**FUJITSU SEMICONDUCTOR**
CONTROLLER MANUAL

CM81-00205-2E

# FR-V/FR FAMILY

# SOFTUNE C/C++ COMPILER MANUAL

## for V5

FUJITSU

# FR-V/FR FAMILY

# SOFTUNE C/C++ COMPILER MANUAL

**for V5**

**FUJITSU LIMITED**

# PREFACE

## ■ Objective of This Manual and Target Readers

This manual describes the Sofutune C/C++ compiler (hereinafter referred to as the C/C++ compiler) usage procedures and libraries.

This manual is prepared for persons who use the above-mentioned compiler and create and development application programs in C and C++ language. Read this manual thoroughly before starting.

This manual is to be read by persons who have a basic knowledge of each MCU (Micro Controller Unit).

The compiler described in this manual conforms about C language to the American National Standard for Information Systems Programming Language C, X3.159-1989, which is abbreviated ANSI standard in this manual. Part of "ISO/IEC 14882:1998" is used to explain C++.

## ■ Trademarks

Softune is a trademark of Fujitsu Ltd.

FR stands for FUJITSU RISC Controller developed by Fujitsu Ltd.

Microsoft, Windows, Windows NT, MS-DOS are registered trademarks of Microsoft Corporation in the USA and/or other countries.-DOS is a registered trademark of Microsoft Corp.

UNIX is a registered trademark that X/Open Co., Ltd. has licensed in the United States and other countries.

Other trademarks or registered trademarks are the property of their respective owners.

The ™ or ® mark is not used within this manual.

## ■ Structure of This Manual

This manual consists of 11 chapters and an Appendix:

**CHAPTER 1 "SOFTUNE C/C++ COMPILER"**

This chapter gives a general description of the C/C++ compiler.

**CHAPTER 2 "SETTING ENVIRONMENT VARIABLES IN SYSTEM BEFORE STARTING"**

This chapter describes the environment variables in the system for operating the C/C++ compiler.

**CHAPTER 3 "C/C++ COMPILER OPERATION"**

This chapter explains the functional specifications of commands.

**CHAPTER 4 "fcc911s COMMAND OBJECT PROGRAM STRUCTURE"**

This chapter explains the requirements for FR family program execution.

**CHAPTER 5 "fcc935s COMMAND OBJECT PROGRAM STRUCTURE"**

This chapter explains the requirements for FR-V family program execution.

**CHAPTER 6 "EXTENDED LANGUAGE SPECIFICATIONS"**

This chapter deals with the extended language specifications supported by the compiler. It also explains the restrictions on translation by the compiler.

**CHAPTER 7 "EXECUTION ENVIRONMENT"**

This chapter relates to the procedures for execution in the environment without an operating system.

**CHAPTER 8 "LIBRARY OVERVIEW"**

The chapter outlines the library on the configuration of files supplied by the library and the relationship between the library and the system where the library is installed.

**CHAPTER 9 "LIBRARY INCORPORATION"**

This chapter discusses the operations and functions required for use of the library.

**CHAPTER10 "COMPILER-DEPENDENT SPECIFICATIONS"**

This chapter describes the specifications depending on the compiler. The description is related to the JIS standards based on the ANSI standards.

**CHAPTER 11 "SIMULATOR DEBUGGER LOW-LEVEL FUNCTION LIBRARY"**

This chapter explains how to use the low-level function library for a simulator debugger.

**APPENDIX**

The Appendix lists the types, macros and functions supplied by the library and covers operations that are unique to the library.

## ■ Grammer Books

For C language syntax and standard library functions, refer to commercially available standard compliant reference books.

## ■ Reference Books


THE C PROGRAMMING LANGUAGE
(Brian W.Kernighan & Dennis M.Ritchie)

Japanese edition entitled Programming Language C UNIX Type Programming Method and Procedure
(Translated by Haruhisa Ishida; Kyoritsu Shuppan)

American National Standard for Information Systems - Programming Language
C, X3.159-1989

UNIX system User’s Manual system V
(Western Electric Company, CIncorporated)

UNIX system V Programmer Reference Manual
(AT&T Bell Laboratories)

User Reference Manual UTS/5 Release 0.1
(Western Electric Company, Incorporated and Amdahl Corporation)

UTS Command Reference Manual UTS/5 Release 0.1
(Western Electric Company, Incorporated and Amdahl Corporation)

The Annotated Reference Manual
(Addison-Wesley Publishing Company,Inc.)

The Programing Language C++ Third Edition
(Addison-Wesley Publishing Company,Inc.)

ISO/IEC 14882:1998

1. The contents of this document are subject to change without notice. Customers are advised to consult with FUJITSU sales representatives before ordering.
2. The information and circuit diagrams in this document are presented as examples of semiconductor device applications, and are not intended to be incorporated in devices for actual use. Also, FUJITSU is unable to assume responsibility for infringement of any patent rights or other rights of third parties arising from the use of this information or circuit diagrams.
3. The contents of this document may not be reproduced or copied without the permission of FUJITSU, Ltd.
4. FUJITSU semiconductor devices are intended for use in standard applications (computers, office automation and other office equipment, industrial, communications, and measurement equipment, personal or household devices, etc.).
   ***CAUTION:***
   *Customers considering the use of our products in special applications where failure or abnormal operation may directly affect human lives or cause physical injury or property damage, or where extremely high levels of reliability are demanded (such as aerospace systems, atomic energy controls, sea floor repeaters, vehicle operating controls, medical devices for life support, etc.) are requested to consult with FUJITSU sales representatives before such use. The company will not be responsible for damages arising from such use without prior approval.*
5. Any semiconductor devices have an inherent chance of failure. You must protect against injury, damage or loss from such failures by incorporating safety design measures into your facility and equipment such as redundancy, fire protection, and prevention of over-current levels and other abnormal operating conditions.
6. If any products described in this document represent goods or technologies subject to certain restrictions on export under the Foreign Exchange and Foreign Trade Law of Japan, the prior authorization by Japanese government will be required for export of those products from Japan.

©2001 FUJITSU LIMITED Printed in Japan

# How To Read This Manual

## ■ Symbols in This Manual

This manual uses the following symbols:

| Symbol | Meaning |
|---|---|
| [item] | Items can be omitted. |
| {item 1 \| item 2} | One of either item 1 or item 2must be specified. The same also applies to teh case where theree or more items are specified. |
| item... | Items can be specified any number of times in succession. |

Most example in this manual are base on the UNIX OS format.

## ■ Abbreviated Name

Each product name in this manual is abbreviated as follows:

Window 95 for Microsoft® Windows® 95 operating system

Windows 98 for Microsoft® Windows® 98 operating system

Windows NT for Microsoft® Windows NT® Server network operating system Version 3.51, Microsoft® Windows NT® Workstation operating system Version 3.51, Microsoft® Windows NT® Server network operating system Version 4.0, Microsoft® Windows NT® Workstation operating system Version 4.0

Windows for both Windows 95 , Windows 98 and Windows NT

MS-DOS for Microsoft MS-DOS® operating system Version 6.2

# CONTENTS


**CHAPTER 1 SOFTUNE C/C++ COMPILER ..... 1**

1.1 Softune C/C++ Compiler Functions ..... 2
1.2 Basic Process of Commands ..... 3
1.3 C/C++ Compiler Basic Functions ..... 5

**CHAPTER 2 SETTING ENVIRONMENT VARIABLES IN SYSTEM BEFORE STARTING . 7**

2.1 FETOOL ..... 8
2.2 LIB911/LIB935 ..... 9
2.3 OPT911/OPT935 ..... 10
2.4 INC911/INC935 ..... 11
2.5 TMP ..... 12
2.6 FELANG ..... 13

**CHAPTER 3 C/C++ COMPILER OPERATION ..... 15**

3.1 Command Line ..... 16
3.2 Command Operands ..... 17
3.3 File Names and Directry Names ..... 18
3.4 Command Options ..... 19
3.4.1 List of Command Options ..... 20
3.4.2 List of Command Cancel Options ..... 24
3.5 Details of Options ..... 27
3.5.1 Translation Control Related Options ..... 28
3.5.2 Preprocessing Related Options ..... 31
3.5.3 Data Output Related Options ..... 34
3.5.4 Language Specification Related Options ..... 40
3.5.5 Optimization Related Options ..... 46
3.5.6 Output Object Related Options ..... 56
3.5.7 Debug Information Related Options ..... 63
3.5.8 Command Related Options ..... 64
3.5.9 Linkage Related Options ..... 65
3.5.10 Option File Related Options ..... 67
3.6 Option Files ..... 68
3.7 Messages Generated in Translation Process ..... 70

**CHAPTER 4 fcc911s COMMAND OBJECT PROGRAM STRUCTURE ..... 73**

4.1 Section Structure of fcc911s ..... 74
4.2 Rules for Name Generation with the fcc911s ..... 76
4.3 fcc911s Command Boundary Alignment ..... 77
4.4 fcc911s Command Bit Field ..... 78
4.5 fcc911s Command Structure/Union ..... 80
4.6 fcc911s Command Function Call Interface ..... 82
4.6.1 fcc911s Command Stack Frame ..... 83
4.6.2 fcc911s Command Argument ..... 85

4.6.3 fcc911s Command Argument Extension Format ..... 88
4.6.4 fcc911s Command Calling Procedure ..... 89
4.6.5 fcc911s Command Register ..... 91
4.6.6 fcc911s Command Return Value ..... 92
4.7 fcc911s Command Interrupt Function Call Interface ..... 93
4.7.1 fcc911s Command Interrupt Stack Frame ..... 94
4.7.2 fcc911s Command Interrupt Function Calling Procedure ..... 95

**CHAPTER 5 fcc935s COMMAND OBJECT PROGRAM STRUCTURE ..... 97**
5.1 Section Structure of fcc935s ..... 98
5.2 Generation Rules for Names Used by fcc935s ..... 100
5.3 fcc935s Command Boundary Alignment ..... 101
5.4 fcc935s Command Bit Field ..... 102
5.5 fcc935s Command Structure/Union ..... 104
5.6 fcc935s Command Function Call Interface ..... 106
5.6.1 fcc935s Command Stack Frame ..... 107
5.6.2 fcc935s Command Arguments ..... 109
5.6.3 fcc935s Command Argument Extension Format ..... 111
5.6.4 fcc935s Command Calling Procedure ..... 112
5.6.5 fcc935s Command Register ..... 114
5.6.6 fcc935s Command Return Value ..... 116

**CHAPTER 6 EXTENDED LANGUAGE SPECIFICATIONS ..... 117**
6.1 Assembler Description Functions ..... 118
6.2 Interrupt Control Functions ..... 124
6.3 I/O Area Access Function ..... 127
6.4 In-line Expansion Specifying Function ..... 129
6.5 Section Name Change Function ..... 130
6.6 Interrupt Level Setup Function ..... 131
6.7 Branch Probability Specification Function ..... 132
6.8 Built-in Function ..... 133
6.8.1 Media Instruction Built-in Function ..... 135
6.8.2 Integer Operation Built-in Function ..... 150
6.9 Predefined Macros ..... 151
6.10 Limitations on Compiler Translation ..... 153
6.11 Re-include Prevention Function ..... 155
6.12 Function for Controlling Instantiation of C++ Template ..... 156

**CHAPTER 7 EXECUTION ENVIRONMENT ..... 157**
7.1 Execution Process Overview ..... 158
7.2 Startup Routine Creation ..... 160

**CHAPTER 8 LIBRARY OVERVIEW ..... 163**
8.1 File Organization ..... 164
8.2 Relationship to Library Incorporating System ..... 166

**CHAPTER 9 LIBRARY INCORPORATION ..... 167**
9.1 Library Incorporation Overview ..... 168

9.2 Initialization/Termination Process Necessary for Library Use ..... 169
9.3 Low-level Function Types ..... 170
9.4 Standard Library Functions and Required Processes/Low-level Functions ..... 171
9.5 Low-level Function Specifications ..... 172
9.5.1 open Function ..... 173
9.5.2 close Function ..... 174
9.5.3 read Function ..... 175
9.5.4 write Function ..... 176
9.5.5 lseek Function ..... 177
9.5.6 isatty Function ..... 178
9.5.7 sbrk Function ..... 179
9.5.8 __exit Function ..... 180
9.5.9 __abort Function ..... 181

**CHAPTER 10 COMPILER-DEPENDENT SPECIFICATIONS ..... 183**
10.1 Compiler-dependent C Language Specification Differentials ..... 184
10.2 Type of Floating-point Data and Range of Representable Values ..... 186
10.3 Dissimilarities between C++ Specifications for C/C++ Compiler and ISO ..... 187
10.4 C++ Specifications for C/C++ Compiler and EC++ Specifications ..... 188
10.5 Limitations on Use of C++ Template ..... 189

**CHAPTER 11 SIMULATOR DEBUGGER LOW-LEVEL FUNCTION LIBRARY ..... 191**
11.1 Low-level Function Library Overview ..... 192
11.2 Low-level Function Library Use ..... 193
11.3 Low-level Func. Function ..... 195
11.4 Low-level Function Library Change ..... 197

**APPENDIX ..... 199**
APPENDIX A List of Types, Macros, Functions, and Variables Provided by C Libraries ..... 200
APPENDIX B Operations Specific to C Libraries ..... 206

**INDEX .....211**

# CHAPTER 1 SOFTUNE C/C++ COMPILER

---

**This chapter outlines the C/C++ compiler. The C/C++ compiler (hereinafter referred to as the compiler) is a language processor program which translates source programs written in C or C++ language into the assembly language for Fujitsu-provided various microcontroller units or microprocessor units.**

---


1.1 "Softune C/C++ Compiler Functions"

1.2 "Basic Process of Commands"

1.3 "C/C++ Compiler Basic Functions"

# 1.1 Softune C/C++ Compiler Functions

**The C/C++ compiler generates an assembler source file from a C or C++ source file.**

## ■ C/C++ Compiler Functions

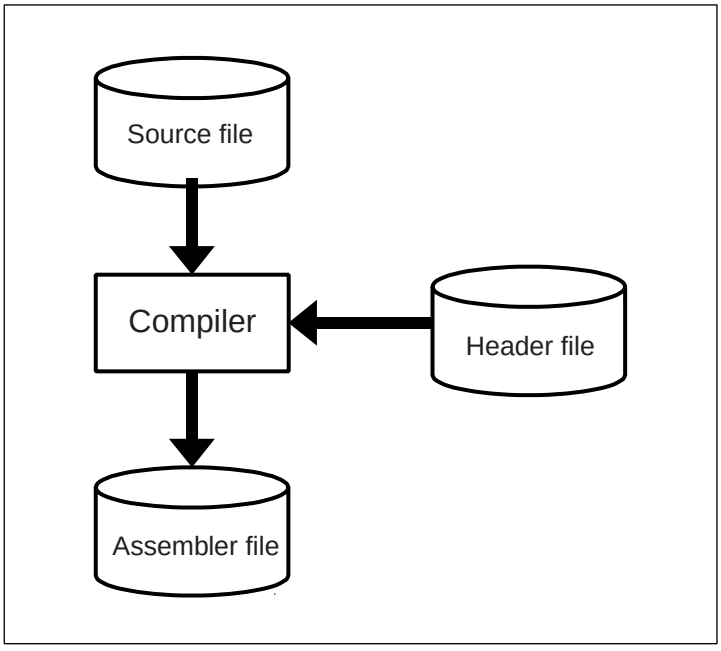
The C/C++ compiler generates an assembler source file using the procedures described below.

### ❍ [Compilation]

Compiling is performed by the compiler (cpcoms). The compiler translates C or C++ source files into assembler source files.

To operate the C/C++ compiler, use the fcc911s or fcc935s command. This command automatically sets the environment of the C/C++ compiler (such as the setting of an include directory) and controls compiling of C or C++ source files. Figure 1.1-1 "Configuration of the C/C++ compiler" shows the configuration of the C/C++ compiler.

**Figure 1.1-1 Configuration of the C/C++ compiler**




In the subsequent sections, the C/C++ compiler translation process is explained using commands. For the details of the command function specifications, see CHAPTER 3 "C/C++ COMPILER OPERATION".

# 1.2 Basic Process of Commands

**This section describes the basic function of commands used in the C/C++ compiler.**
**The C/C++ compiler uses the following command:**

- **fcc911s ----- FR family command**
- **fcc935s ----- FR-V family command**

## ■ Command Basic Process

The basic function of the command is to generate an absolute file from a C or C++ source file. The command recognizes files with the .c extension as C source files, and files with the .cc, .cpp, or .cxx extension as C++ source files.

A command use example is given below. > is the command prompt.

**[Example]**

```
> fcc911s -cpu MB91100 file.c
> fcc935s -cpu MB93500 file.c
```

When the above entry is made, the command assumes that file.c is a C source file. As far as no error is detected, an absolute file (file.abs) is generated in the current directory.

**[Example]**

```
> fcc911s -o outfile -cpu MB91100 file.c
> fcc935s -o outfile -cpu MB93500 file.c
```

With the parameters set as indicated in the above example, the command generates an absolute file "outfile." By specifying options such as -o, the command can control the file generation process.

## ■ Options for Compiling Process Control

### ❍ [-P option]

When the -P option is specified, the command calls up the compiler only and performs preprocessing to generate a preprocessed C/C++ source file in the current directory. Files to be generated include files with extensions changed to .i for C and with extensions changed to .ipp for C++.

### ❍ [-S option]

When the -S option is specified, the command calls up the compiler to compile and thus generate an assembler source file in the current directory. The extension of the generated file is changed to .asm.

### ❍ [-c option]

When the -c option is specified, the command calls up the, compiler, and assembler and performs compilation, and assembling to generate an object file in the current directory. The extension of the generated file is changed to .obj.

- **[-o option]**

  When the -o option is specified, the command generates the file specified in the command line as a result of processing.

# 1.3 C/C++ Compiler Basic Functions

**The C/C++ compiler three functions are described below.**

- **Header file search**
- **Coordination with symbolic debugger**
- **Optimization**

**The symbolic debugger is a support tool for analyzing a program created in C language or C++ language.**

## ■ Header File Search

The header file can be acquired using the C or C++ program #include instruction. When the absolute pathname is specified, the header file enclosed within angular brackets (<>) is searched for in the directory defined by that pathname. When the absolute pathname is not specified, the compiler standard directory is searched.

The standard header file is supplied by the C/C++ compiler.

The header file enclosed by double quotation marks (") is searched for in a directory specified by the absolute pathname. If the absolute pathname is not specified, such a header file is searched for in a directory having a file containing a #include line. If the header file is not found in a directory having a file containing a #include line, the standard directory is searched next.

The -I option makes it possible to add a directory for header file search.

**[Example]**

```
> fcc911s -cpu MB91100 -I ..\include file.c
> fcc935s -cpu MB93500 -I ..\include file.c
```

When the above entry is made, the command searches for the header file enclosed within angular brackets in the order indicated below.

1. ../include
2. Compiler standard include file directory

The header file enclosed by double quotation marks is searched for in the order indicated below.

1. Current directory having a file containing a #include line
2. ../include
3. Compiler standard include file directory

The -I option can be specified a desired number of times. When it is specified two or more times, search operations are conducted in the specified order.

## ■ Coordination with Symbolic Debugger

When the -g option is specified, the compiler generates the debug information to be used by the symbolic debugger. When such information is generated, C/C++ language level debugging can be accomplished within the symbolic debugger. Two types of symbol debuggers are available; simulator debugger and emulator debugger.

When the optimization option (-O[1-4]) is specified, debugging should be performed, noting the

following points. When the optimization option is specified, the compiler attempts to ensure good code generation by changing the computation target position and eliminating computations that are judged to be unnecessary. To minimize the amount of data exchange with memory, the compiler tries to retain data within a register. It is therefore conceivable that a break point positioned in a certain line may fail to cause a break or that currently monitored certain address data may fail to vary with the expected timing. It also well to remember that the debug data will not be generated for an unused local variable or a local variable whose area need not be positioned in a stack as a result of optimization.

Debugging must be conducted with the above considerations taken into account.

### ■ Optimization

When the -O option is specified, the compiler generates an object subjected to general-purpose optimization.

# CHAPTER 2 SETTING ENVIRONMENT VARIABLES IN SYSTEM BEFORE STARTING

---

**This chapter describes environment variables in the system used to run the C/C++ compiler. (For information on setting variables, refer to the manual for the respective operating system.) All environment variables can be omitted. For information on the supply style, refer to the C/C++ Compiler Installation Manual.**
**See Section 3.3 "File Names and Directry Names", for details about the characters that can be used for the directories to be set up as environment variables in the Windows95/Windows98/WindowsNT version.**
**[Setup Example]**

```
set TMP=c:\Fujitsu MCU tool
```

**For environment variable setup, do not use double quotation marks (").**

---


2.1 "FETOOL"

2.2 "LIB911/LIB935"

2.3 "OPT911/OPT935"

2.4 "INC911/INC935"

2.5 "TMP"

2.6 "FELANG"

# 2.1 FETOOL

## Specify the directory where the development environment is installed.

### ■ FETOOL

**[General Format 1 For UNIX OS]**

setenv FETOOL Installation directory

**[General Format 2 For Windows]**

set FETOOL = Installation directory

The driver accesses the compiler, message file, include file, and other items via the path specified by FETOOL.

When FETOOL setup is not completed, the parent directory for the directory where the activated driver exists (the /.. position of the directory where the driver exists) is regarded as the installation directory.

No more than one directory can be specified.

**[Example For UNIX OS]**

setenv FETOOL /usr/local/softune5

**[Example For Windows]**

set FETOOL=c:\softune5

# 2.2 LIB911/LIB935

**Specify the directory that contains the library to which the fcc911s commands are linked by default as LIB911. Specify the directory that contains the library to which the fcc935s commands are linked by default as LIB935.**

## ■ LIB911/LIB935

**[General Format 1 For UNIX OS]**

setenv LIB911 library directory [:directory 2...]

setenv LIB935 library directory [:directory 2...]

**[General Format 2 For Windows]**

set LIB911 = library directory [; directory 2...]

set LIB935 = library directory [; directory 2...]

Specify the directory to which linking is effected by default.

When LIB911 and LIB935 are not set up, the directory placed at the respective location relative to the FETOOL directory (%FETOOL%\lib\911 or %FETOOL%\lib\935) is regarded as the default library directory.

Two or more directories can be specified, separated by a delimiter. The delimiter is a colon ':' for the UNIX OS, and a semicolon ';' for Windows.

**[Example For UNIX OS]**

setenv LIB911 /usr/local/softune5/lib/911

setenv LIB935 /usr/local/softune5/lib/935

**[Example For Windows]**

set LIB911=c:\softune5\lib\911

set LIB935=c:\softune5\lib\935

# 2.3 OPT911/OPT935

**Specify the directory for the default option file to be used by the fcc911s command as OPT911. Specify the directory for the default option file to be used by the fcc935s command as OPT935.**

## ■ OPT911/OPT935

**[General Format 1 For UNIX OS]**

setenv OPT911 Default option file directory

setenv OPT935 Default option file directory

**[General Format 2 For Windows]**

set OPT911 = Default option file directory

set OPT935 = Default option file directory

Specify the directory for the default option file to be used by the driver.

When OPT911 and OPT935 are not set up, the directory at the corresponding relative position with respect to the FETOOL directory (%FETOOL%\lib\911 or %FETOOL%\lib\935) is regarded as the default option file directory.

No more than one directory can be specified.

**[Example For UNIX OS]**

setenv OPT911 /usr/local/softune5/lib/911

setenv OPT935 /usr/local/softune5/lib/935

**[Example For Windows]**

set OPT911=c:\softune5\lib\911

set OPT935=c:\softune5\lib\935

# 2.4 INC911/INC935

**Specify the directory where fcc911s commands search for standard header files as INC911. Specify the directory where fcc935s commands search for standard header files as INC935.**

## ■ INC911/INC935

**[General Format 1 For UNIX OS]**

setenv INC911 Standard include directory

setenv INC935 Standard include directory

**[General Format 2 For Windows]**

set INC911 = Standard include directory

set INC935 = Standard include directory

Specify the directory where the compiler searches for standard header files. The directory specified as INC911 or INC935 is regarded as the standard include-directory. When INC911 or INC935 is not defined, the directory at the corresponding relative position with respect to the FETOOL directory (%FETOOL%\lib\911\include or %FETOOL%\lib\935\include) is regarded as the standard header file directory.

Two or more directories can be specified, separated by a delimiter. The delimiter is a colon ':' for the UNIX OS, and a semicolon ';' for Windows.

**[Example For UNIX OS]**

setenv INC911 /usr/local/softune5/lib/911/include

setenv INC935 /usr/local/softune5/lib/935/include

**[Example For Windows]**

set INC911=c:\softune5\lib\911\include

set INC935=c:\softune5\lib\935\include

# 2.5 TMP

## Specify the directory for the temporary file to be used by the C compiler.

### ■ TMP

**[General Format 1 For UNIX OS]**

setenv TMP Temporary directory

**[General Format 2 For Windows]**

set TMP = Temporary directory

Specify the working directory for creating the temporary file to be used by the C compiler.

If TMP setup is not completed, the temporary file is created in the /tmp directory for UNIX OS or in the current directory for Windows.

No more than one directory can be specified.

**[Example For UNIX OS]**

setenv TMP /usr/tmp

**[Example For Windows]**

set TMP=c:\tmp

# 2.6 FELANG

## Specify the code for messages.

### ■ FELANG

**[General Format 1 For UNIX OS]**

setemv FELANG Message code

**[General Format 2 For Windows]**

set FELANG = Message code

Specify the message code. The following codes can be specified.

**ASCII:**

Outputs messages in ASCII code. The generated messages are in English. Select this code for a system without a Japanese language environment.

**EUC:**

Outputs messages in EUC code. The generated messages are in Japanese.

**SJIS:**

Outputs messages in SHIFT JIS code. The generated messages are in Japanese. If FELANG setup is not completed, the ASCII code is considered to be selected.

**[Example For UNIX OS]**

setenv FELANG EUC

**[Example For Windows]**

set FELANG=SJIS

# CHAPTER 3 C/C++ COMPILER OPERATION

**This chapter describes the command function specifications.**


3.1 "Command Line"

3.2 "Command Operands"

3.3 "File Names and Directry Names"

3.4 "Command Options"

3.5 "Details of Options"

3.6 "Option Files"

3.7 "Messages Generated in Translation Process"

# 3.1 Command Line

**The command line format is shown below.**

- **fcc911s [options] operands**
- **fcc935s [options] operands**

## ■ Command Line

Options and operands can be specified in the command line. Options and operands can be specified at any position within the command line. Two or more options and operands can be specified. Options can be omitted.

Option and operand entries are to be delimited by a blank character string. The command recognizes the options and operands in the order indicated below.

1. An entry beginning with a hyphen (-) is first recognized as an option. The subsequent character string is interpreted to determine the option type.
2. As regards an option having an argument, the subsequent character string is regarded as the argument.
3. The remaining entries in the command line are recognized as operands.

**[Example]**

```
>fcc911s file1.c -S -I \home\myincs file2.c -cpu MB91100
>fcc935s file1.c -S -I \home\myincs file2.c -cpu MB93500
```

At first, -S and -I are regarded as options. Since the -I option has an argument, the subsequent character string /home/myincs is regarded as the argument. The remaining entries (file1.c and file2.c) are regarded as operands.

**Options:**

-S, -I \home\myincs

**Operands:**

file1.c, file2.c

## ■ Command Process

The command calls a compiler, assembler, and linker for each input file in the order they are specified to compile, assemble, and link. The result of each module is output to a file having the same file name as an input file name with its extension replaced by .obj. The result after linking is output to a file having the same file name with its extension replaced by .abs.

The linking result, unless changed by the -o option, is output to files with .abs extensions to which the extensions of the files first specified are changed.

**[Example]**

```
>fcc911s file1.c file2.c file3.c -cpu MB91100
>fcc935s file1.c file2.c file3.c -cpu MB93500
```

Files named file1.c, file2.c, and file3.c are subjected to compiling and assembling so that files named file1.obj, file2.obj, and file3.obj are generated.

# 3.2 Command Operands

## One or more input files can be specified as operands.

### ■ Command Operands

The command determines the file type according to the input file extension and performs an appropriate process to suit the file type. The extension cannot be omitted.

**File specifying:**

C source files, preprocessed C source files, C++ source files, preprocessed C++ source files, assembler source files, and object files.

**File extension:**

The relationship between input file extensions andcommand processes is shown in Table 3.2-1 "Relationship between Extensions and Command Processes".

**Table 3.2-1 Relationship between Extensions and Command Processes**

| Extension | Command Process |
|---|---|
| .c | Files having this extension are compiled and subjected to subsequent processes as C source files. |
| .i | Files having this extension are compiled and subjected to subsequent processes as C source files that have been preprocessed. |
| .cc | Regard files as the C++ source files to perform processing subsequent to compiling. |
| .cpp | Regard files as the C++ source files to perform processing subsequent to compiling. |
| .cxx | Regard files as the C++ source files to perform processing subsequent to compiling. |
| .ipp | Regard files as the preprocessed C++ source files to perform processing subsequent to compiling. |
| .asm | Files having this extension are assembled and subjected to subsequent processes as assembler source files that have been compiled. |
| .obj | Files having this extension are assembled and subjected to subsequent processes as object files that have been assembled. |
| .abs | The file having this extension is regarded as a linked absolute file, and an error output is generated. No absolute file can be specified. |

Note, however, that the associated process may be inhibited depending on the option specifying

**[Example]**

```
>fcc911s file1.c file2.asm -cpu MB91100
>fcc935s file1.c file2.asm -cpu MB93500
```

The file1.c file is compiled and assembled, and the file2.asm file is assembled. Linking is then performed to generate the file1.abs file.

# 3.3 File Names and Directry Names

**The following characters are applicable to file names and directory names.**

## ■ File Names and Directry Names

### ❍ Windows

Alphanumeric characters, symbols except \, /, :, *, ?, ", <, >, and |, Shift-JIS kanji codes, and Shift- JIS 1-byte kana codes. Enclose a long file name in double quotation marks (") when the file is specified as an option or operand. The double quotation marks, however, cannot be used when the file is specified as an environment variable.

### ❍ UNIX OS

Underbar (_) and alphanumeric characters (However, the first character must be the underbar or alphabetical character).

### ❍ Module name

A module name is generated from a file name. The characters that can be used for a module name are an underbar "_" and alphanumeric characters. (The first character of the name must be an underbar "_" and an alphabetic character.) When other characters are used for a file name, a character that cannot be used for a module name is converted to an underbar "_." A file name that will be identical to a module name after conversion should not be used.

**[Example]**

#abc.c and abc.c will have the same module name (_abc).

# 3.4 Command Options

## The command options are explained below.

### ■ Option Syntax

The option consists of a hyphen (-) and one or more characters following the hyphen. Some options have an argument. A blank character string must be positioned between an option and an argument. The command options cannot be grouped for purposes of specifying. Grouping is a technique of specifying which, for instance, uses a -Sg form to specify both the -S option and -g option.

### ■ Multiple Specifying of Same Option

If the same option is specified more than one time, only the last-specified option in the command line is assumed to be valid.

**[Example]**

```
>fcc911s -o outfile file.c -o outobj -cpu MB91100
>fcc935s -o outfile file.c -o outobj -cpu MB93500
```

The name of a file to be output is outobj. The options listed below can be specified more than once for the same command and are different in meaning each time.

Options that are significant when specified more than one time

-D, -f, -I, -INF, -K, -L, -l, -ra, -ro, -sc, -T, -U, -x, -Y

When the above options are specified more than one time, see details of options.

### ■ Position within Command Line

The option's position within the command line does not have a special meaning. Options are interpreted in the same manner no matter where in the command line they are specified.

**[Example]**

1. 
```
>fcc911s -C -E file1.c file2.c -cpu MB91100
>fcc935s -C -E file1.c file2.c -cpu MB93500
```
2. 
```
>fcc911s file1.c -E file2.c -C -cpu MB91100
>fcc935s file1.c -E file2.c -C -cpu MB93500
```

The same processing operations are performed for cases 1 and 2.

### ■ Exclusiveness and Dependency

Options may be mutually exclusive or mutually dependent. For information on the exclusivity and dependence of options, see option descriptions.

### ■ Case Sensitiveness

As regards the options, their upper-case and lower-case characters are different from each other. For example, the -O option is different from the -o option. However, the upper- and lower-case characters of suboptions are not differentiated from each other. For example, the -K eopt option is considered in the same as the -K EOPT option. The suboptions are the character strings that follow the -K option or -INF option.

## 3.4.1 List of Command Options

**When executed without argument specifying, the command outputs an option list to the standard output.**
**Table 3.4-1 "List of Command Options" to Table 3.4-5 "List of Command Cancel Options" list the command options. Options listed in the tables are recognized by the command.**

■ **List of Command Options**

**Table 3.4-1 List of Command Options**

| Specifying Format | Function |
|---|---|
| -B | Allow a C++-style comment //. |
| -C | Leaves a comment in the preprocessing result. |
| -c | Perform processing up to assembling and output the result to .obj. |
| -cmsg | Outputs the compiling process end message to the standard output. |
| -cpu MB number | Specify the MB number of the CPU used. |
| -cwno | Set the end code for warning to 1. |
| -D name [= [tokens]] | Defines the macro name. |
| -E | Performs preprocessing only and outputs the result to the standard output. |
| -f filename | Specifies the option file. |
| -g | Adds to the object the information necessary for debugging. |
| -H | Outputs the acquired header file pathname to the standard output. |
| -help | Outputs the option list to the standard output. |
| -I dir | Specifies the directory for head file search. |
| -INF LIST | Generates the assemble list. |
| -INF LONGMESSAGE | Details a message at translating |
| -INF {SRCIN \| LINENO} | Insert the associated C source information as a comment into the assembler source. |
| -INF STACK [=filename] | Generates the stack use amount data. |
| -J {a \| c \| e} | Specifies the specification level of the language to be interpreted by the compiler. |

**Table 3.4-1 List of Command Options (Continued)**

| Specifying Format | Function |
|---|---|
| -K {DCONST \| FCONST} | Specifies the type of a real constant without a suffix. |
| -K EOPT | Effects optimization for changing the arithmetic operation evaluation procedure. |
| -K LIB | Recognizes the standard function operation and implements in-line expansion/substitution for other functions. |
| -K NOALIAS | Effects optimization on the presumption that differing pointers do not indicate the same area. |
| -K NOINTLIB | Effects no in-line expansion for interrupt related functions. |
| -K NOUNROLL | Inhibits loop unrolling. |
| -K NOVOLATILE | Does not consider __io qualifier variables to be volatile. |
| -K {SIZE \| SPEED} | Selects optimization based on the size and execution speed. |
| -K {UCHAR\|SCHAR} | Specifies the mere char sign handling. |
| -K {UBIT\|SBIT} | Specifies the mere int bit field sign handling. |
| -kanji{SJIS \| EUC} | Specify the kanji code used in a program. |
| -O level | Gives instructions for general-purpose optimization. |
| -o pathname | Outputs the result to the pathname. |
| -P | Performs preprocessing only and outputs the result to *.i. |
| -S | Performs processes up to compiling and outputs the result to *.asm. |
| -s defname = newname [, attr [, address]] | Changes the section name. |
| -T item, arg1 [arg2...] | Passes arguments to the tool. |
| -U name | Cancels the macro name definition. |
| -V | Outputs the executed compiler tool version information to the standard output. |
| -varorder{SORT\|NORMAL} | Specify the allocation type of static variables. |
| -w level | Specifies the warning message output level. |
| -Xdof | Inhibits the default option file read operation. |
| -x func [, func2...] | Specifies the in-line expansion of functions. |
| -xauto [size] | Specifies the in-line expansion of the functions whose logical line count is not less than size. |
| -Y item, dir | Changes the item position to dir |

**Table 3.4-1 List of Command Options (Continued)**

| Specifying Format | Function |
|---|---|
| -e name | Specifies the entry of a program. |
| -L path1 [,path2...] | Specifies the library path. |
| -l lib1 [, lib2...] | Specifies the library file name. |
| -m | Outputs a map file at the time of linking. |
| -ra name = start/end | Specifies the RAM area. |
| -ro name = start/end | Specifies the ROM area. |
| -sc param | Specifies the section arrangement. |
| -startup file | Specifies the startup file name. |

**Table 3.4-2 List of Command Options (For fcc911s)**

| Specification format | Function |
|---|---|
| -K {A1 \| A4} | Specifies the minimum boundary alignment value for static data. |
| -K REALOG | Performs in-line expansion for the ITRON system call function. |
| -K {SCHEDULE \| NOSCHEDULE} | Specifies the recall of the scheduler. |
| -K {SARG \| DARG} | Specifies the method for argument area acquisition. |
| -K {SHORTADDRESS [= {CODE \| DATA} ] \| LONGADDRESS [= {CODE \| DATA} ] } | Specifies the method for handling external symbols. |

**Table 3.4-3 List of Command Options (For fcc935s)**

| Specification format | Function |
|---|---|
| -align {FUNC8 \| FUNC16 \| LABEL8 \| LABEL16 \| LOOP8 \| LOOP16 \| INNERLOOP8 \| INNERLOOP16} | Specifies that branch labels are to be aligned. |
| -K {A1 \| A4 \| A8} | Specifies the minimum boundary alignment value for static data. |
| -K GS | Specifies that global scheduling is to be applied for instruction transfer beyond the branch. |
| -K NOSCHEDULE | Specifies that optimization for instruction scheduling is not effected. |
| -K PREDICATE | Specifies that a conditional instruction is output. |
| -K (NEO \| NE1 \| NE2) | Specifies the output method for the Non-excepting instruction. |

**Table 3.4-3 List of Command Options (For fcc935s) (Continued)**

| Specification format | Function |
|---|---|
| -K NOSWPL | Specifies that software pipelining is not performed. |
| -splitsection | Specifies that CODE sections having different names are generated for different functions. |

**Table 3.4-4 List of Command Options (for C++ Source)**

| Specifying Format | Function |
|---|---|
| -t {none \| used \| all \| local} | Specify the type of template instantiation. |
| -alternative_tokens | Enable an alternative keyword. |
| -no_auto_instrantiation | Suppress automatic instantiation of a template. |
| -old_for _init | Use the ANSI or earlier specifications for the scope of declaration in the for initialize expression. |
| -for_init diff_warning | Control a warning on different interpretations between --old_for_init and --new_for_init. |
| -suppress_vtbl | Suppress definition of a virtual function table. |
| -force_vtbl | Force definition of a virtual function table. |

## 3.4.2 List of Command Cancel Options

**The listed options are used to cancel command options on an individual basis. The cancel options for the command are listed in Table 3.4-6 "List of Command Cancel Options (For fcc911s)" to Table 3.4-7 "List of Command Cancel Options (For fcc935s)".**

■ **List of Command Cancel Options**

**Table 3.4-5 List of Command Cancel Options**

| Specifying Format | Function |
|---|---|
| -XB | Cancel the -B option. |
| -XC | Cancels the -C option. |
| -Xcmsg | Cancels the -cmsg option. |
| -Xcwno | Cancel the -cwno option. |
| -Xf | Cancels the -f option. |
| -Xg | Cancels the -g option. |
| -XH | Cancels the -H option. |
| -Xhelp | Cancels the -help option. |
| -XI | Cancels the -I option. |
| -INF NOLINENO | Cancels the LINENO suboption. |
| -INF NOLIST | Cancels the LIST suboption. |
| -INF NOSRCIN | Cancels the SRCIN suboption. |
| -INF NOSTACK | Cancels the STACK suboption. |
| -INF SHORTMESSAGE | Cancel the -LONGMESSAGE suboption. |
| -K ALIAS | Cancels the NOALIAS suboption. |
| -K INTLIB | Cancels the NOINTLIB suboption. |
| -K NOEOPT | Cancels the EOPT suboption. |
| -K NOLIB | Cancels the LIB suboption. |
| -K UNROLL | Cancels the NOUNROLL suboption. |
| -K VOLATILE | Cancels the NOVOLATILE suboption. |
| -Xo | Cancels the -o option. |
| -Xs | Cancels the -s option. |
| -XT item | Cancels the T item specifying. |

**Table 3.4-5 List of Command Cancel Options (Continued)**

| Specifying Format | Function |
|---|---|
| -XV | Cancels the -V option. |
| -Xx | Cancels the -x option. |
| -Xxauto | Cancels the -xauto option. |
| -XY item | Cancels the Y item specifying. |
| -Xe | Cancels the -e option. |
| -XL | Cancels the -L option. |
| -Xl | Cancels the -l option. |
| -Xm | Cancels the -m option. |
| -Xra | Cancels the -ra option. |
| -Xro | Cancels the -ro option. |
| -Xsc | Cancels the -sc option. |
| -Xstartup | Cancels the -startup option. |

**Table 3.4-6 List of Command Cancel Options (For fcc911s)**

| Specification format | Function |
|---|---|
| -K NOREALOS | Cancels the REALOS suboption. |

**Table 3.4-7 List of Command Cancel Options (For fcc935s)**

| Specification format | Function |
|---|---|
| -Xalign | Cancels the -align option. |
| -K NOGS | Cancels the -K GS option. |
| -K SCHEDULE | Cancels the -K NOSCHEDULE option. |
| -K NOPREDICATE | Cancels the -K PREDICATE option. |
| -K SWPL | Cancels the -K NOSWPL option. |
| -Xsplitsection | Cancels the -splitsection option. |

**Table 3.4-8 List of Command Cancel Options (For C++ Source)**

| Specifying Format | Function |
|---|---|
| --no_alternative_tokens | Cancel the --alternative_tokens option. |
| --auto_instantiation | Cancel the --no_auto_instantiation option. |
| --new_for_int | Cancel the --old_for_init option. |
| --no_for_int_diff_warning | Cancel the --for_init_diff_warning option. |

# 3.5 Details of Options

## This section details the options.

- **Translation control related options**

  The translation control related options are related to preprocessing, compiler, assembler, and linker call control.

- **Preprocessor related options**

  The preprocessor related options are related to preprocessing operations

- **Data output related options**

  The data output related options are related to the command, preprocessing, and compiler data outputs.

- **Language specification related options**

  The language specification related options are related to the specification of the language to be recognized by the compiler.

- **Optimization related options**

  The optimization related options are related to the optimization to be effected by the compiler.

- **Output object related options**

  The output object related options are related to the output object format.

- **Debug information related options**

  The debug information related options are related to the debug information to be referenced by the symbolic debugger.

- **Command related options**

  The command related options are related to the other tools recalled by commands.

- **Linkage related options**

  The linkage related options are related to linkage.

- **Option file related options**

  The option file related options are associated with option files.

# 3.5.1 Translation Control Related Options

**The translation control related options are related to preprocessor, compiler, assembler, and linker call control.**

## ■ Translation Control Related Options

The priorities of the translation control related options are defined as follows. They are not related to the order of specifying.

-E > -P > -S > -c

The translation control related option exclusiveness is shown in Table 3.5-1 "Translation Control Related Option Exclusiveness"

**Table 3.5-1 Translation Control Related Option Exclusiveness**

| Specified Option | Option Invalidated |
|---|---|
| -E | -S and -c |
| -P | -S and -c |
| -S | -c |
| -c | None |

If the -E and -P options are specified simultaneously, see the explanation below.

### ❍ -E Option

-E Option is subjects all files to preprocessing and outputs the result to the standard output. The output result contains the preprocessing instruction generated by the preprocessor, which is necessary for the compiler. The information targets for the preprocessing instruction generated by the preprocessor are the #line and #pragma instructions. If the -P option is specified together with the -E option, the preprocessing instruction generated by the preprocessor is inhibited.

If the input file is not a C source file or C++ source file, the -E option does not do anything.

**[Example]**

```
>fcc911s -E -cpu MB91100 sample.c
>fcc935s -E -cpu MB93500 sample.c
```

The sample.c preprocessing result is output to the standard output.

### ❍ -P Option

-P option subjects a C source file or C++ source file to preprocessing and outputs the result to the file whose extension is changed to .i or ipp. Unlike the cases where the -E option is specified, the output result does not contain the preprocessing instruction generated by the preprocessing. If the input file is not a C source file, the -P option does not do anything.

**[Example]**

```
>fcc911s -P -cpu MB91100 sample.c
>fcc935s -P -cpu MB93500 sample.c
```

The sample.c preprocessing result is output to the sample.i.

**[Example]**

```
>fcc911s -P  -cpu  MB91100  sample.cpp
>fcc935s -P  -cpu  MB93500  sample.cpp
```

The result of preprocessing for sample.cpp is output to sample.ipp.

❍ **-S Option**

-S option performs processes up to compiling and outputs the resultant assembler source to file extension changed to .asm.

If the input is neither a C source file nor a preprocessed C source file, the -S option does not do anything.

**[Example]**

```
>fcc911s -S  -cpu  MB91100  sample.c
>fcc935s -S  -cpu  MB93500  sample.c
```

The sample.c preprocessing and compiling process result are output to the sample.asm.

❍ **-c Option**

Performs processes up to assembling and outputs the resultant object to file extension changed to .obj. If the input file is an object file, the -c option does not do anything.

**[Example]**

```
>fcc911s  -c  -cpu  MB91100  sample.c
>fcc935s  -c  -cpu  MB93500  sample.c
```

The results of preprocessing, compiling, and assembling to sample.c are output to sample.obj.

The relationship among file types, translation control related options, and processes is shown in Table 3.5-2 "Relationship Among File Types, Translation Control Related Options, and Processes".

**Table 3.5-2 Relationship Among File Types, Translation Control Related Options, and Processes**

| Option File Type | -E | -P | -S | -c | Nothing Specified |
|---|---|---|---|---|---|
| C/C++ source file | P | P | P and C | P, C and A | P, C, A and L |
| Preprocessed C/C++ source file | - | - | C | C and A | C, A and L |
| Assembler source file | - | - | - | A | A and L |
| Object file | - | - | - | - | L |

P: Preprocessing
C: Compiling
A: Assembling
L: Linking

**[Example]**

```
>fcc911s  -E  file1.c  file2.i  -cpu  MB91100
>fcc935s  -E  file1.c  file2.i  -cpu  MB93500
```

Subjects files named file1.c to preprocessing and outputs the result to the standard output. No processing is performed for file2.i.

```
>fcc911s -S file1.c file2.i file3.asm -cpu MB91100
>fcc935s -S file1.c file2.i file3.asm -cpu MB93500
```

Subjects a file named file1.c to preprocessing and compiling and a file named file2.i to compiling. Performs nothing for a file named file3.asm. As a result, files named file1.asm and file2.asm are generated in the current directory.

## 3.5.2 Preprocessing Related Options

**This section deals with the options related to preprocessing operations. If the preprocessing is not called, the preprocessing related options are invalid.**

### ■ Preprocessing Related Options

The preprocessing related options are detailed below.

#### ❍ -B option and -XB option

The -B option regards items following // in the C source as comments.

The -XB option cancels the -B option. This option is ignored even if it is specified in the C source.

**[Example]**

```
input:      void func( )
            {
                // empty function
            }
operation:  fcc911s  -S  -B  -cpu  MB91100  sample.c
            fcc935s  -S  -B  -cpu  MB93500  sample.c
```

#### ❍ -C option and -XC option

-C options are All comments except those which are in the preprocessing instruction line will be retained as the preprocessing result. If the option is not specified, the comments are replaced by one blank character.

The -XC option cancels the -C option.

**[Output Example]**

```
Input:      /* Comment */
            void func(void) { }
Operation:  fcc911s  -C  -E  -cpu  MB91100  sample.c
            fcc935s  -C  -E  -cpu  MB93500  sample.c
Output:     # 1 "test5.c"
            /* Comment */
            void func(void) { }
```

#### ❍ -D name [=[tokens]] Option

Defines the macro name with the tokens used as the macro definition. This option is equal to the following #define instruction.

#define name tokens

If =tokens entry is omitted, the value 1 is given as the tokens value. If the tokens entry is omitted, the specified lexis is deleted from the source file. The error related to the -D option is the same as the error related to the #define instruction.

This option can be specified more than one time.

**[Example]**

```
>fcc911s -D os=m -D sys file.c -cpu MB91100
>fcc935s -D os=m -D sys file.c -cpu MB93500
```

In a file named file.c, processing is conducted on the assumption that the macro definitions for os and sys are m and 1, respectively.

❍ **-H option and -XH option**

-H option outputs to the standard output the header file pathnames acquired during preprocessing. The pathnames are sequentially output, one for each line, in the order of acquisition. If there are any two exactly the same pathnames, only the first one will be output. When this option is specified, the command internally sets up the -E option to subjects all files to preprocessing only. However, the preprocessing result will not be output. The -XH option cancels the -H option.

**[Output Example]**

```
Input:      #include <stdio.h>
            #include "head.h"
Operation:  fcc911s -H -cpu MB91100 sample.c
Output:     /usr/softune5/lib/911/include/stdio.h
            ./head.h
Operation:  fcc935s -H -cpu MB93500 sample.c
Output:     /usr/softune5/lib/935/include/stdio.h
            ./head.h
```

❍ **-I dir option and -XI option**

Changes the manner of header file search so that the directory specified by dir will be searched prior to the standard directory.

The standard directory is ${INC911} for the fcc911s command, and ${INC935} for the fcc935s command

This option can be specified more than one time. The search will be conducted in the order of specifying. When this option is specified, the header file search will be conducted in the following directories in the order indicated below.

**[Header file enclosed within angular brackets (< >)]**

1. Directory specified by the -I option
2. Standard directory

**[Header file enclosed by double quotation marks (")]**

1. Directory having a file containing the #include line
2. Directory specified by the -I option
3. Standard directory

If a header file is specified by specifying its absolute path name, only the directory specified by the specified absolute path name will be searched. If any nonexistent directory is specified, this option is invalid.

The -XI option cancels the -I option.

❍ **-U name option**

Cancels the macro name definition formulated by -D. This option is equivalent to the following #undef instruction.

#undef name

If the same name is specified by the -D and -U options, the name definition will be canceled without regard to the order of option specifying. This option can be specified more than one time.

The error related to the -U option is the same as the error related to the #undef instruction.

**[Example]**

```
>fcc911s -U m -D n -D m file.c -cpu MB91100
>fcc935s -U m -D n -D m file.c -cpu MB93500
```

This will cancel the macro m definition formulated by the -D option.

# 3.5.3 Data Output Related Options

**This section deals with the options related to the command, preprocessor, and compiler data outputs.**

## ■ Data Output Related Options

### ❍ -cmsg Option

Outputs the compiling process completion message.

**[Example]**

```
Operation:  fcc911s -cmsg -S -cpu MB91100 sample.c
            fcc935s -cmsg -S -cpu MB93500 sample.c
Output:     COMPLETED C Compile CFOUND NO ERROR : sample.c
```

### ❍ -cwno Option

Set the end code to 1 when a warning-level error occurs. When this option is not given, the end code is 0.

### ❍ -help option and -Xhelp option

Outputs the option list to the standard output.

The -Xhelp option cancels the -help option.

**[Example]**

```
>fcc911s -help
>fcc935s -help
```

Various command option lists are output to the standard output.

### ❍ -INF LINENO option and -INF NOLINENO option

-INF LINENO option inserts C or C++ source file line numbers into the assembler source file as comments. The LINENO suboption cannot be specified simultaneously with the SRCIN suboption.

The NOLINENO suboption cancels the LINENO suboption.

**[Output Example]**

```
Input:      void    func(void){ }

Operation:  fcc911s  -INF     lineno    -S  -cpu  MB91100  sample.c
Output:     _func:
                    ST          RP,     @-SP
                    ENTER       #4
            ;;;;        sample.c    line 1
            L_func:
                    LEAVE
                    LD          @SP,    RP
                    RET

Operation:  fcc935s  -INF     lineno    -S  -cpu  MB93500  sample.c  -K    noschedule
Output:     _func:
            L_20:
            ;;;;    optl.c,     line 1
                    subi        sp, #16, sp
                    st          fp, @(sp, gr0)
                    mov         sp, fp
                    movsg       LR, gr4
                    sti         gr4, @(fp, 8)
            L_21:
            L_22:
                    ldi         @(fp, 8), gr4
                    movgs       gr4, LR
                    addi        fp, #16, sp
                    ld          @(fp, gr0), fp
                    ret
```

❍ **-INF LIST option and -INF NOLIST option**

-INF LIST option outputs the assemble list. Generates a file in the current directory. The name of the generated file is determined by changing the source file name extension to .lst.

Since the assemble list is generated at assembling, it is not generated when assembling is not conducted.

For the details of the assemble list, refer to the Assembler Manual.

The NOLIST suboption cancels the LIST suboption.

**[Example]**

```
>fcc911s  -INF list  -c  -cpu  MB91100  sample.c

>fcc935s  -INF list  -c  -cpu  MB93500  sample.c
```

The sample.c preprocessing, compiling, and assembling process result are output to the sample.obj, and the resulting assemble list is output to the sample.lst.

❍ **-INF SRCIN option and -INF NOSRCIN option**

-INF SRCIN option inserts a C or C++ source file into the assembler source file as a comment. The NOSRCIN suboption cancels the SRCIN suboption.

The SRCIN suboption cannot be specified simultaneously with the LINENO suboption.

**[Output Example]**

```
Input:      void    func(void) { }

Operation:  fcc911s  -INF     srcin     -S  -cpu  MB91100  sample.c
Output:     _func:
                     ST       RP,    @-SP
                     ENTER    #4
            ;;;;      void    func(void){ }
            L_func:
                     LEAVE
                     LD       @SP+,  RP
                     RET

Operation:  fcc935s  -INF     srcin     -S  -cpu  MB93500  sample.c  -K  noschedule
Output:     _func:
            L_20:
            ;;;;     void  func(void) { }
                     subi     sp, #16, sp
                     st       fp, @(sp, gr0)
                     mov      sp, fp
                     movsg    LR, gr4
                     sti      gr4, @(fp, 8)
            L_21:
            L_22:
                     ldi      @(fp, 8), gr4
                     movgs    gr4, LR
                     addi     fp, #16, sp
                     ld       @(fp, gr0), fp
                     ret
```

❍ **-INF STACK [=file]option and -INF NOSTACK option**

-INF STACK [=file]option outputs the stack use amount data. Generates the specified file in the current directory and outputs the stack use amount data. If no file is specified, the information in all the simultaneously compiled files is output into files whose names are determined by changing the source file extensions to .stk.

For stack use amount data utilization procedures and data file specifications, refer to the Softune C/C++analyzer Manual.

The NOSTACK suboption cancels the STACK suboption.

**[Output Example]**

```
Input:      extern  void  sub(void);
            void    func(void) {sub( );}

Operation:  fcc911s  -INF  stack  -S  -cpu  MB91100  sample.c
Output:     @sample.c
            #     E=Extern    S=Static    I=Interrupt
            #       {Stack}     {E|S|I}     {function name}  [A]
            #            ->     {E|S}       {call function}
            #                   ...
            #
                        8       E           _func
                        ->      E           _sub

Operation:  fcc935s  -INF  stack  -S  -cpu  MB93500  sample.c
Output:
            #     E=Extern    S=Static    I=Interrupt
            #       {Stack}     {E|S|I}     {function name}  [A]
            #            ->     {E|S}       {call function}
            #                   ...
            #
                        40      E           _func
                        ->      E           _sub
```

❍ **-o pathname option and -Xo option**

-o pathname option uses the pathname as the output file name. If this option is not specified, the default for the employed file format is complied with.

The -Xo option cancels the -o option.

**[Example]**

```
>fcc911s  -o  output.asm  -S  -cpu  MB91100  sample.c
>fcc935s  -o  output.asm  -S  -cpu  MB93500  sample.c
```

The sample.c preprocessing and compiling process result are output to the output.asm.

❍ **-V option and -XV option**

-V option outputs the version information about each executed compiler tool to the standard output. The - XV option cancels the -V option.

**[fcc911 Command Output Example]**

```
FR Family Softune C/C++ Compiler V50L01

ALL RIGHTS RESERVED, COPYRIGHT (C) FUJITSU LIMITED 1998

LICENSED MATERIAL - PROGRAM PROPERTY OF FUJITSU LIMITED
```

**[fcc935 Command Output Example]**

```
FR-V Family Softune C/C++Compiler V50L01
```

ALL RIGHTS RESERVED, COPYRIGHT (C)FUJITSU LIMITED 1986

LICENSED MATERIAL -PROGRAM PROPERTY OF FUJITSU LIMITED

❍ **-w [level] option**

-w [level] option specifies the output level of warning-type diagnostic messages. The level can be specified between 0 and 8 but there is no difference between levels 1 and 8. This level specification is provided to ensure compatibility between the Softune C compiler and option.

When level 0 is specified, no warning messages will be generated. If the level description is omitted, level 0 is assumed. When the -w [level] option itself is omitted, -w 1 applied.

For the details of diagnostic messages, see section 3.7 "Messages Generated in Translation Process".

**[Output Example]**

```
Input:      const unsigned int a=-1;_
Operation:  fcc911s -w 5 -S -cpu MB91100 sample.c
            fcc935s -w 5 -S -cpu MB93500 sample.c
Output:     *** sample.c(1) W1068B: warning: integer conversion resulted in a change of sign
Operation:  fcc911s -w -S -cpu MB91100 sample.c
            fcc935s -w -S -cpu MB93500 sample.c
Output:     (None)
```

❍ **--for_init_diff_warning option and --no_for_init_diff_warning option**

The --for_init_diff_warning option is valid only when the input file is a C++ source file. This option is ignored when the --old_for_init option is specified. The --for_init_diff_warning option specifies that a warning message is output if the initialize statement in the for statement is interpreted differently between the old and new C++ specifications.

The --no_for_init_diff_warning option cancels the --for_init_diff_warning option.

**[Output Example]**

```
Input:
    void func(int);
    int i;
    main( )
    {
      for(int  i=0;  i<10;  i++)
          func(i);
      func(i);
    }
Operation:  fcc911s -S -cpu MB91100 --for_init_diff_warning sample.cpp
            fcc935s -S -cpu MB93500 --for_init_diff_warning sample.cpp
Output:     *** sample. cpp(7) W1780B: warning: reference is to variable "i" (declared at line 2)
             -- under old for-init scoping rules it would have been variable "i"(declared at line 5)
```

### ❍ -INF LONGMESSAGE option and -INF SHORTMESSAGE option

The -INF LONGMESSAGE option details the diagnostic message of the compiler. The '^' mark indicates where an error occurs. Note that an incorrect location may be indicated depending the error type. It should also be noted that if the diagnostic message is displayed in a proportional font, the indicated location seems displaced.

The -INF SHORTMESSAGE option cancels the -INF LONGMESSAGE option.

**[Example]**

```
Input:
      int a b;
      void func( )
      {
      }
Operation:  fcc911s -S -cpu MB91100 -INF LONGMESSAGE sample.c
            fcc935s -S -cpu MB93500 -INF LONGMESSAGE sample.c
Output:     *** sample.c(1) E4065B: expected a ";"
              int a b;
                    ^
```

## 3.5.4 Language Specification Related Options

**This section deals with the options related to the specifications of the language to be recognized by the compiler.**

### ■ Language Specification Related Options

#### ❍ -J {a | c | e} +Option

Specifies the language specification level to be interpreted by the compiler (preprocessor included).

When -Ja is specified, interpretation is conducted in compliance with the ANSI specifications including expansion specifications.

When -Jc is specified, interpretation is conducted in strict compliance with the ANSI specifications. In response to the expansion specifications, a warning message is output.

When the -Je option is specified, the C++ source file is interpreted as being based on the EC++ language specifications. In addition to a normal warning, a warning message is output at description of specifications outside the range of the EC++ language specifications. The C source file has exactly the same meaning as that when the -Ja option is specified.

If omit -J {a|c|e}, this option is not specified, -Ja applies.

**[Example]**

```
>fcc911s -J a file1.c -J c file2.c  -cpu  MB91100

>fcc935s -J a file1.c -J c file2.c  -cpu  MB93500
```

The -Jc option becomes valid so that files named file1.c and file2.c are interpreted in strict compliance with the ANSI specifications.

#### ❍ -K {DCONST|FCONST}Option

When the FCONST suboption is specified, a floating-point constant whose suffix is not specified will be handled as a float type. When the DCONST suboption is specified, a floating-point constant whose suffix is not specified will be handled as a double type. If neither of the above two suboptions is specified, -K DCONST applies.

**[Output Example]**

```
Input:      extern  float  f1, f2;
            void  func(void) { f1 = f2+1.0;}

Operation:  fcc911s  -K  fconst  -cpu  MB91100  -S  sample.c
Output:     _func:
                    ST       RP,  @-SP
                    ENTER #4
                    LDI:32  #_f2,     R12
                    LD       @R12,   R4
                    LDI       #H' 3F800000,  R5
                    CALL32 _ _addf,  R12
                    LDI:32  #_f1,  R12
                    ST       R4,  @R12
            L_func:
                    LEAVE
                    LD       @SP+,  RP
                    RET

Operation:  fcc935s -K  fconst  -cpu  MB93500 -S  sample.c  -K  noschedule
Output:     _func:
            L_20:
                    subi        sp, #16, sp
                    st          fp, @(sp, gr0)
                    mov         sp, fp
                    movsg       LR, gr4
                    sti         gr4, @(fp, 8)
            L_21:
                    sethi       #hi(_f2), gr4
                    setlo       #lo(_f2), gr4
                    ldf         @(gr4, gr0), fr1
                    sethi       #hi(1065353216), gr5
                    setlo       #lo(1065353216), gr5
                    movgf       gr5, fr2
                    fadds       fr1, fr2, fr3
                    sethi       #hi(_f1), gr6
                    setlo       #lo(_f1), gr6
                    stf         fr3, @(gr6, gr0)
            L_22:
                    ldi         @(fp, 8), gr4
                    movgs       gr4, LR
                    addi        fp, #16, sp
                    ld          @(fp, gr0), fp
                    ret
```

❍ **-K NOINTLIB option and -K INTLIB option**

The NOINTLIB suboption calls a normal function without effecting in-line expansion of an interrupt related function (__DI(), __EI(), __set_il()). The INTLIB suboption cancels the NOINTLIB suboption.

Interrupt-related functions can be used only for fcc911s commands.

**[Example]**

```
Input:      void  func(void){ _ _DI( );}

Operation:  fcc911s -K nointlib -cpu MB91100 -S sample.c
Output:     _func:
                ST      RP, @-SP
                ENTER   #4
                CALL32  _ _ _DI, R12
            L_func:
                LEAVE
                LD      @SP+, RP
                RET
```

❍ **-K NOVOLATILE option and -K VOLATILE option**

The NOVOLATILE suboption does not recognize a __io qualifier attached variable as a volatile type. Therefore, __io qualifier attached variables will be optimized. The VOLATILE suboption cancels the NOVOLATILE suboption.

**[Example]**

```
>fcc911s -K novolatile -S -O -cpu MB91100 sample.c

>fcc935s -K novolatile -S -O -cpu MB93500 sample.c
```

When an __io qualifier attached variable is processed in sample.c, it is not handled as a __volatile qualifier attached variable, but is treated as the optimization target.

❍ **-K {UCHAR|SCHAR}option**

-K {UCHAR|SCHAR}option specifies whether or not to treat the char type most significant bit as a sign bit. When the UCHAR suboption is specified, the most significant bit will not be treated as a sign bit. When the SCHAR suboption is specified, the most significant bit will be treated as a sign bit.

If omit -K {UCHAR|SCHAR}option neither of the above two suboptions is specified, -K UCHAR applies.

**[Output Example]**

```
Input:      extern   int          data;
            char                   c = -1;
            void   func(void) { data = c;}

Operation:  fcc911s  -K schar -cpu MB91100 -S sample.c
Output:             LDI:32    #_c, R12
                    LDUB      @R12, R0
                    EXTSB     R0                    ; Code-extended
                    LDI:32    #_data, R12
                    ST        R0, @R12

Operation:  fcc935s  -K schar -cpu MB93500 -S sample.c -K noschedule
Output:             sethi     #hi(_c), gr4
                    setlo     #lo(_c), gr4
                    ldsb      @(gr4, gr0), gr5      ; Code-extended
                    sethi     #hi(_data), gr6
                    setlo     #lo(_data), gr6
                    st        gr5, @(gr6, gr0)
```

❍ **-K REALOS option and -K NOREALOS option**

The REALOS suboption effects in-line expansion of the ITRON system call function.

It can be used in cases where a program running under REALOS is to be prepared. This option can only be used in the fcc911s command. For the ITRON system call function, refer to the REALOS/FR Kernel Manual.

When specifying the REALOS suboption, be sure to include the system call declaration header file furnished by the REALOS.

If the REALOS suboption is specified without including the system call declaration header file and system call inline expansion is initiated, the operation is not guaranteed, because it is possible that an adequate argument-type check has not been completed.

The NOREALOS suboption cancels the REALOS suboption.

**[Output Example]**

```
Input:      #include   "itron.h
            #include   "realos.h"
            void  func(void) { ext_tsk( ); }
Operation:  fcc911s  -K realos -cpu MB91100 -S sample.c
Output:     LDI:8      #-21, R12
            EXTSB      R12
            INT        #64
```

❍ **-K {UBIT|SBIT} option**

Specifies whether or not to treat the most significant bit as a sign bit in situations where the char, short int, or long int type is selected as the bit field. When the UBIT suboption is specified, the most significant bit will not be treated as a sign bit. When the SBIT suboption is specified, the most significant bit will be treated as a sign bit.

If neither of the above two suboptions is specified, -K UBIT applies.

**[Output Example]**

```
Input:      extern int    data;
            struct tag{ int bf:1;}st = {-1};
            void func(void) { data = st.bf;}

Operation: fcc911s -K sbit -cpu MB91100 -S sample.c
Output:            LDI:32   #_st, R12
                   LDUB     @R12, R0
                   EXTSB    R0                  ; Code-extended
                   ASR      #7, R0
                   LDI:32   #_data, R12
                   ST       R0, @R12

Operation: fcc935s -K sbit -cpu MB93500 -S sample.c -K noschedule
Output:            sethi    #hi(_st), gr4
                   setlo    #lo(_st), gr4
                   ld       @(gr4,gr0), gr5
                   srai     gr5, #31, gr6       ; Code-extended
                   sethi    #hi(_data), gr7
                   setlo    #lo(_data), gr7
                   st       gr6, @(gr7, gr0)
```

❍ **-kanji {SJIS|EUC} option**

If Japanese words are entered in a program, the code system for the used Japanese words is specified.

Japanese words including half-size kana can be entered in program comments and character strings. The compiler identifies the code system for Japanese description based on this option. SJIS means that the Shift JIS code system is used, and EUC means that the EUC code system is used. When this option is omitted, -kanji EUC is used for Solaris, and -kanji SJIS is used for HP-UX and Windows.

❍ **--alternative_tokens option and --no_alternative_tokens option**

The --alternative_tokens option enables alternative representation. For the specifications for alternative representation, refer to ISO/ICE 14882:1988.

The --no_alternative_tokens option cancels the --alternative_tokens option.

❍ **--old_for_init option and --new_for_init option**

The --old_for_init option makes a change so that the scope of variables declared in the initialize statement in the for statement is interpreted as being within the old specifications.

The --new_for_init option cancels the --old_for_init option.

**[Example]**

Input:

```
extern void func(int);
int main( )
{
   for(int i=0; i<10; i++)
        func(i);
   func(i);                    // Possible only for old specifications
  return 0;
}
```

Operation: fcc911s -S --old_for_init sample.cpp -cpu MB91100
fcc935s -S --old_for_init sample.cpp -cpu MB93500

# 3.5.5 Optimization Related Options

**This section deals with the options related to optimization by the compiler.**

## ■ Optimization Related Options

### ❍ -K SIZE option

Selects an appropriate optimization combination with emphasis placed upon the object size. The selected options in the fcc911s command are indicated below.

- -O 3
- -K EOPT
- -K NOUNROLL
- -K SHORTADDRESS

The options for the fcc935s command are indicated below.

- -O 3
- -K EOPT
- -K NOUNROLL

To change the above combination, specify, for example, the 01 option following the -SIZE sub-option.

The -K SIZE option not only offers the optimization combination selection function, but also makes it possible to issue a generation instruction for object size minimization and effect object pattern switching.

### ❍ -K SPEED option

Selects an appropriate optimization combination with emphasis placed upon the generated object execution speed. The selected options in the fcc911s command are indicated below.

- -O 4
- -K SHORTADDRESS

The option for the fcc935s command is indicated below.

- -O 4

To change the above combination, specify, for example, the -K LONGADDRESS option following the -SIZE sub-option.

The -K SPEED option not only offers the optimization combination selection function, but also makes it possible to issue a generation instruction for execution speed maximization and effect object pattern switching.

### ❍ -O [level]option

-O [level] specifies the optimization level. Levels 0, 1, 2, 3, and 4 can be specified. The higher the optimization level, the shorter the generated object execution time but the longer the compilation time. Note that each optimization level contains lower optimization level functions.

One of the following levels is to be specified. When no level is specified, -O2 applies.

- O(Optimization level 0): No optimization will be effected. This level is equivalent to cases where the -O is not specified.
- 1(Optimization level 1): Optimization will be effected in accordance with detailed analyses of a program flow.
- 2(Optimization level 2): The following optimization feature is exercised in addition to the feature provided by optimization level 1.
  - Loop unrolling: Loop unrolling is performed to increase the execution speed by decreasing the loop count when loop- count detection is possible. However, it tends to increase object size. Therefore, this optimization should not be used in situations where object size is important.

```
(Before unrolling)
        for(i=0;i<3;i++) { a[i]=0;}
(After unrolling)
        a[0]=0;
        a[1]=0;
        a[2]=0;
```

- 3(Optimization level 3): The following optimization features are exercised in addition to the features provided by optimization level 2.
  - Loop unrolling (extended): Loops, including branch instructions, that have not been the target of optimization level-2 loop unrolling, are the target of this extended loop unrolling.
  - Optimization function repeated execution: In optimization function repeated execution, the optimization features except the loop unrolling feature will be repeatedly executed until no more optimization is needed. However, the translation time will increase.
- 4(Optimization level 4): The following optimization features are exercised in addition to the features provided by optimization level 3.
  - Arithmetic operation evaluation type change (same as effected by -K EOPT specifying): Switches to a higher-speed standard function that recognizes standard function operations, performs standard function inline expansion, and performs identical operations. When this option is specified, there may be side effects on the execution results. Since standard function inline expansion is implemented, the code size may increase. Performs optimization to change arithmetic operation evaluation type at compilation stage. When this option is specified, there may be side effects on the execution results.
  - Standard function expansion/change (same as effected by -K LIB specifying): Switches to a higher-speed standard function that recognizes standard function operations, performs standard function inline expansion, and performs identical operations. When this option is specified, there may be side effects on the execution results. Since standard function inline expansion is implemented, the code size may increase.

❍ **-K EOPT option and -K NOEOPT option**

The EOPT suboption effects optimization by changing the arithmetic operation evaluation type at the compilation stage. When this option is specified, side effects may occasionally be produced on the execution results. This option takes effect only when an optimization level of 1 or higher is specified by the -O option. The NOEOPT suboption cancels the EOPT suboption.

**[Output Example]**

```
Input:      extern int i;
            void func(int a, int b) {
              i=a-100+b+100;
            }

Operation: fcc911s -K eopt -O -cpu MB91100 -S sample.c
Output:             ADD  R5, R4              ; Order of arthmatic operation replaced
                    LDI:32 #_i, R12
                    ST     R4, @R12

Operation: fcc935s -K eopt -O -cpu MB93500 -S sample.c -K noschedule
                    add    gr10, gr11, gr4   ; Order of arthmatic operation replaced
                    sethi  #hi(_i), gr5
                    setlo  #lo(_i), gr5
                    st     gr4, @(gr5, gr0)
```

❍ **-K LIB option and -K NOLIB option**

The LIB suboption recognizes the standard function operation and replaces the standard function with a higher-speed standard function which effects standard function in-line expansion and performs the same operation as the original standard function. When this option is specified, side effects may be produced on the execution results. Since standard function inline expansion is implemented, the code size may increase.

This option takes effect only when an optimization level of 1 or higher is specified by the -O option.

The NOLIB suboption cancels the LIB suboption.

**[Output Example]**

```
Input:      extern   int i;
            void   func(void) {
              i=strlen("ABC");
            }

Operation:  fcc911s   -K  lib  -O  -cpu  MB91100  -S  sample.c
Output:               LDI     #3, R0              ; Processing equivalent to strlen
                                                    expanded
                      LDI:32  #_i, R12
                      ST      R0, @R12

Input:      extern char s[ ];
            void   func(void) {
              strcpy(s,"ABC");
            }

Operation:  fcc935s   -K  lib  -O  -cpu  MB93500  -S  sample.c  -K  noschedule
Output:               sethi   #hi(_s), gr4
                      setlo   #lo(_s), gr4
                      sethi   #hi(LS_0), gr5
                      setlo   #lo(LS_0), gr5
                      mov     gr4, gr8
                      mov     gr5, gr9
                      setlos  #4, gr10
                      call    __cmov              ; Processing equivalent to strlen
                                                    expanded
```

❍ **-K {LONGADDRESS [= {CODE|DATA}] | SHORTADDRESS [= {CODE|DATA}]}option**

The SHORTADDRESS suboption generates code on the presumption that the (address) of a symbol to be loaded within the program is within the 20-bit expression range.

This option can be specified only for fcc911s commands.

When CODE or DATA is specified, only the program code section (CODE) symbols or data section (DATA, INIT, or CONST) symbols are to be processed. If the address exceeds the range that can be expressed by 20 bits, an error occurs during linking. Operation is normal even if symbols other than those loaded in the program are positioned outside the range of the addresses that can be expressed by 20 bits. The LONGADDRESS suboption enables handling symbol addresses as 32-bit addresses. If the SHORTADDRESS suboption or LONGADDRESS suboption is omitted, -K LONGADDRESS is applied.

**[Output Example]**

```
Input:      extern int i;
            extern void sub(void);
            void func(void) {
              i=10;
              sub( );
            }

Operation:  fcc911s  -K  shortaddress  -O  -S  -cpu  MB91100  sample.c

Output:             LDI:20   #_i, R12          ; 20-bit symbol used
                    LDI      #10, R0
                    ST       R0, @R12
                    CALL20   _sub, R12        ; 20-bit symbol used
```

❍ **-K NOALIAS option and -K ALIAS option**

The NOALIAS suboption optimizes the data specified by the pointer on the assumption that the pointer does not specify the same area as the other variables or pointers.

These options are enabled only when level 1 or more optimization is specified in the -O option. The language specification permits the pointer to point to the same area as any other variable or pointer. Therefore, when using this option, check the program carefully.

The ALIAS suboption cancels the NOALIAS suboption.

**[Output Example]**

```
Input:      extern int  i;
            extern int  j;
            void  func9(int *p) {
              *p=i+1;
               j=i+1;
            }

Operation: fcc911s  -K   noalias  -O  -cpu  MB91100  -S  sample.c
Output:             LDI:32  #_i, R12
                    LD      @R12, R0
                    LDI:32  #_j, R12
                    ADD     #1, R0
                    ST      R0, @R4
                    ST      R0, @R12          ; Value of *p=i+1 reused

Operation: fcc935s  -K   noalias  -O  -cpu  MB93500  -S  sample.c  -K  noschedule
Output:             sethi   #hi(_i), gr4
                    setlo   #lo(_i), gr4
                    ld      @(gr4, gr0), gr5
                    addi    gr5, #1, gr6
                    st      gr6, @(gr10, gr0)
                    sethi   #hi(_j), gr7
                    setlo   #lo(_j), gr7
                    st      gr6, @(gr7, gr0)  ; Value of *p=i+1 reused
```

❍ **-K {SCHEDULE|NOSCHEDULE} option**

-K {SCHEDULE|NOSCHEDULE} option specifies whether or not to implement instruction scheduling. When the SCHEDULE suboption is specified for the fcc911s command, instruction scheduling will be conducted. When the NOSCHEDULE suboption is specified, the command will not conduct instruction scheduling. If this option is omitted, the operation conforms to the contents specified in the -O option. When an optimization level of 1 or higher is specified by the -O option, -K SCHEDULE is assumed to be specified.

When the SCHEDULE suboption is specified for the fcc935s. instruction scheduling will be conducted. When the NON SCHEDULE suboption is specified, the command will not conduct instruction scheduling. This option is only valid when an optimization level of 1 or higher is specified by the -O option. The -K GS option determines whether to perform large-area scheduling, in which instruction that exceed the branch are transferred.

❍ **-K NOUNROLL option and -K UNROLL option**

The NOUNROLL suboption inhibits loop unrolling optimization. Use this option when loop unrolling optimization is to be inhibited with the -O2 to -O4 options specified. The UNROLL suboption cancels the NOUNROLL suboption. When -00 or 01 is specified, these options are invalid because loop unloading is not optimized.

❍ **-x function name 1 [, function name 2,...]option and -Xx option**

The -x option effects in-line expansion, instead of function calling, of functions defined by a C source. However, recursively called functions will not be subjected to in-line expansion. It should also be noted that functions may not be subjected to in-line expansion depending on asm statement use, structure/union type argument presence, setjmp function calling, and other conditions. The -x option takes effect only when level 1 or more optimization is specified simultaneously with the -O option.

The -Xx option cancels the -x option.

**[Output Example]**

```
Input:      extern  int  a;
            static void sub(void) { a=1; }
            void func(void) { sub( ); }

Operation:  fcc911s  -cpu  MB91100  -O  -x  sub  -S  sample.c
Output:     _func:
                    LDI       #1, R0
                    LDI:32    #_a, R12
                    RET:D
                    ST        R0, @R12

Operation:  fcc935s  -cpu  MB93500  -O  -x  sub  -S  sample.c  -K  noschedule
Output:     _func:
                    setlos    #1, gr4
                    sethi     #hi(_a), gr5
                    setlo     #lo(_a), gr5
                    st        gr4, @(gr5, gr0)
                    ret
```

### ❍ -xauto [size]option and -Xxauto option

The -xauto option effects in-line expansion, instead of function calling, of functions whose logical line count is not less than size. However, recursively called functions will not be subjected to in-line expansion. It should also be noted that functions may not be subjected to in-line expansion depending on asm statement use, structure/union type argument presence, setjmp function calling, and other conditions. If the size entry is omitted, the value 30 is assumed to be specified. The values from 1 to 127 are valid for size. The -xauto option takes effect only when level 1 or more optimization is specified simultaneously with the -O option.

The -Xxauto option cancels the -xauto option.

**[Output Example]**

```
Input:      extern  int  a;
            static  void  sub(void) { a=1; }
            void  func(void) { sub( ); }

Operation: fcc911s -cpu  MB91100  -O  -xauto  -S  sample.c
Output:    _func:
                    LDI       #1, R0
                    LDI:32    #_a, R12
                    RET:D
                    ST        R0, @R12

Operation: fcc935s -cpu  MB93500  -O  -xauto  -S  sample.c  -K  noschedule
Output:    _func:
                    setlos    #1, gr4
                    sethi     #hi(_a), gr5
                    setlo     #lo(_a), gr5
                    st        gr4, @(gr5, gr0)
                    ret
```

### ❍ -K GS option and -K NOGS option

The -K GS option specifies that global scheduling is to be applied for instruction transfer beyond the branch.

This option can be used for the fcc935s command only.

When the -K GS option is specified, the compiler searches the most probable route to be executed and performs instruction transfer beyond the branch via this route. However, code is also provided to cancel instruction transfer beyond the branch when the compiler-predicted execution route differs from the actually executed route. The resulting execution speed may be lower than when scheduling beyond the branch is not performed.

When this option is specified, compilation time may become extremely long.

This option takes effect only when an optimization level of 1 or higher is specified simultaneously with the -O option.

The -K GS option is ignored when the source file meets any of the following conditions.

- The source file contains a #pragma asm, #pragma endasm, or an asm statement.
- A single function performs 10,000 or more calculations. Moreover, the number of operations increases or decreases depending on the optimization by the compiler. For example, when loop unrolling is performed, the number of operations increases.
- The source file contains an infinite null loop.

This option is also ignored when the -K NOSCHEDULE option is specified. The NOGS suboption cancels the GS suboption.

**[Output Example]**

Input:
```
extern int a;
void func(int flag) {
   if(flag)   a=1;
   else       a=2;
}
```

Operation: fcc935s -cpu MB93500 -O -K gs -S sample.c

Output:
```
_func:
        add.p     gr0, fp, gr5          ; Global scheduling on the assumption
        subi      sp, #40, sp           ; that if (flag) is established
        sti.p     gr5, @(sp, 0)
        addi      sp, #0, fp
        subcc.p   gr8, gr0, gr0, icc0
        addi      gr0,      #1, gr3
        sethi.p   #HI(_a), gr4
        setlo.p   #LO(_a), gr4
        beq  icc0, #0, L_26
        st        gr3, @(gr4, gr0)
L_27:
        add       fp, gr0, sp
        ldi.p     @(fp, 0), fp
        addi.p    sp, #40, sp
        ret
L_26:
        addi  gr0,      #2, gr5         ; Because this section is processed only when if
        sethi.p   #HI(_a), gr6          ; (flag) is established, the execution speed becomes
        setlo     #LO(_a), gr6          ; slower when if (flag) is not established.
        st        gr5, @(gr6, gr0)
        bra       L_27
```

❍ **-K PREDICATE option and -K NOPREDICATE option**

The -K PREDICATE option specifies that a conditional instruction be output.

This option can be used for fcc935s command only.

When this option is specified, a conditional jump instruction and subsequent instructions are replaced by conditional instructions. Because this eliminates a branch, instruction scheduling may be performed, enabling generation of code with higher execution speed.

However, because the system always executes the instructions on both paths of the conditional jump, the execution speed may become slower than without using conditional instructions.

This option takes effect only when an optimization level of 1 or higher is specified simultaneously with the -O option. The NOPREDICATE suboption cancels the PREDICATE suboption.

**[Output Example]**

```
Input:      extern int a;
            void func(int flag) {
              if(flag) a=1;
            }

Operation: fcc935s -cpu MB93500 -O -K predicate -S sample.c -K noschedule
Output:    _func:
           L_20:
                      subi   sp, #40, sp
                      mov    gr8, gr10
           L_21:
                      cmp    gr10, gr0, icc0
                      setlos #1, gr4
                      sethi  #hi(_a), gr5
                      setlo  #lo(_a), gr5
                      ckeq   icc0, cc4
                      cst    gr4, @(gr5, gr0), cc4, #0
           L_24:
                      addi   sp, #40, sp
                      ret
```

❍ **-K (NE0|NE1|NE2) option**

The -K NEO option specifies that Non-Excepting instructions are not output. The -K NE1 option specifies that a Non-Excepting instruction be output beyond a conditional jump instruction. The -K NE2 option specifies that a Non-Excepting instruction be output beyond a conditional jump instruction and loop.

This option can be used for the fcc935s command only.

Because an exception may occur, replacing the instruction that cannot be transferred beyond the branch with a Non-Excepting instruction can move the instruction beyond the branch. This may allow instruction scheduling, enabling the generation of code with higher execution speed.

However, because the instruction in the path that is not actually executed is transferred beyond the branch, the execution speed may become slower than without a Non-Excepting instruction.

This option takes effect only when an optimization level of 1 or higher is specified simultaneously with the -O option. This option defaults to -K NE0.

**[Output Example]**

```
Input:      extern  int  a, b;
            void func(int flag) {
              if(flag) a=b+1;
              else a=b+2;
            }

Operation: fcc935s    -cpu  MB93500  -O  -K  ne1  -S  sample.c  -K  noschedule
Output:    _func:
           L_20:
                       subi     sp, #40, sp
                       mov      gr8, gr10
           L_21:
                       cmp      gr10, gr0, icc0
                       sethi    #hi(_b), gr10
                       setlo    #lo(_b), gr10
                       nld      @(gr10, gr0), gr11
                       nld      @(gr10, gr0), gr10
                       beq      icc0, #0, L_23

           L_22:
                       clrgr    gr10
                       commitgr         gr11
                       addi     gr11, #1, gr4
                       sethi    #hi(_a), gr5
                       setlo    #lo(_a), gr5
                       st       gr4, @(gr5, gr0)
                       bra      L_25
           L_23:
                       clrgr    gr11
                       commitgr         gr10
                       addi     gr10, #2, gr6
                       sethi    #hi(_a), gr7
                       setlo    #lo(_a), gr7
                       st       gr6, @(gr7, gr0)
           L_25:
                       addi     sp, #40, sp
                       ret
```

❍ **-K NOSWPL option and -K SWPL option**

The -K NOSWPL option specifies that instruction rearrangement based on instruction parallelization is not conducted for optimization of loop unrolling.

This option can be used for the fcc935s command only.

This option takes effect only when an optimization level of 2 or higher is specified simultaneously with the -O option. This option is ignored when the -K NOUNROLL option is specified. The SWPL suboption cancels the NOSWPL suboption.

# 3.5.6 Output Object Related Options

**This section describes the options related to output object formats.**

## ■ Output Object Related Options

### ❍ -align {FUNC8|FUNC16|LABEL8|LABEL16|LOOP8|LOOP16|INNERLOOP8|INNERLOOP16} option and -Xalign option

The -align option aligns branch labels.

This option can only be used with the fcc935s command. The option is valid only when instruction scheduling is performed.

When the FUNC8, LABEL8, LOOP8, or INNERLOOP8 suboption is specified, the compiler suffixes CODE section names with _8, and selects 8 as the section boundary value. When the FUNC16, LABEL16, LOOP16, or INNERLOOP16 suboption is specified, the compiler suffixes CODE section names with _16, and selects 16 as the section boundary value.

When the -s option or #pragma section is used to specify a section arrangement address, it overrides the -align option.

The FUNC8 suboption aligns the beginnings of functions on 8-byte boundaries, while the FUNC16 suboption aligns them on 16-byte boundaries.

The LABEL8 suboption aligns branch labels on 8-byte boundaries, while the LABEL16 suboption aligns them on 16-byte boundaries.

The LOOP8 suboption aligns the beginnings of loops recognized by the compiler on 8-byte boundaries, while the LOOP16 suboption aligns them on 16-byte boundaries.

The INNERLOOP8 suboption aligns the beginnings of innermost loops recognized by the compiler on 8-byte boundaries, while the INNERLOOP16 suboption aligns them on 16-byte boundaries. An innermost loop is the loop at the deepest level in a nested loop structure.

The -Xalign option disables the align value specified for labels. When both -align and -Xalign options are omitted, the -Xalign option is used.

**[Remarks]**

With the FR500, VLIW instructions are fetched on 16-byte boundaries, in units of 16 bytes. When a branch occurs, the first VLIW instruction at the branch destination is fetched. If this VLIW instruction is allocated across a 16-byte boundary, however, the instruction needs to be fetched twice, generating an extra fetch cycle.

The -align option can be used to avoid this extra fetch cycle.

Note that branch labels aligned on boundaries increase the code size, thus possibly reducing the hit rate of the instruction cache.

When a program that uses only integer arithmetic operations is to be compiled, aligning VLIW instructions on 8-byte boundaries may prevent them from crossing 16-byte boundaries. This is because most of them are paired in parallel. If an increase in the code size becomes a bottleneck, specify 8-byte boundary alignment.

**[Output Example]**

```
Input:      void foo(int a, int b, int c) {
                for(; a<b ; a--)
                  a*=c;
            }

Operation: fcc935s -cpu MB93500 -O -K gs -S sample.c -align loop16
Output:    .section        CODE_16, CODE, align=16        ; The section name is changed.
           .global_foo
           .align 16

_foo:
           subcc.p gr8, gr9, gr0, icc1
                   fnop. p                                ; nop is inserted to achieve 16-byte
                                nop                         boundary alignment.
           bge     icc1, #0, L_35
L_36:
           smul    gr8, gr10, gr4
           subi    gr5, #1, gr8
           subcc   gr8, gr9, gr0, icc0
           blt     icc0, #3, L_36
L_35:
           ret
```

❍ **-cpu MB number option**

Specifies MB number of CPU to be used. This option cannot be omitted.

**[Example]**

```
>fcc911s -S -cpu MB91100 sample.c

>fcc935s -S -cpu MB93500 sample.c
```

❍ **-s defname = newname [, attr [, address]] option and -Xs option**

Changes the compiler output section name from defname to newname, and changes section type to attr. An arrangement address can be specified in the address option.

For compiler output section names, see section 4.1 "Section Structure of fcc911s". For selectable section types, refer to the Assembler Manual.

If the arrangement address is specified, the arrangement address cannot be specified relative to the associated section at linking.

The -Xs option cancels the -s option.

**[Output Example]**

```
Input:      void func(void) { }

Operation:  fcc911s  -s CODE=PROGRAM, CODE, 0x1000 -S -cpu MB91100 sample.c
Output:              .SECTION        PROGRAM, CODE, LOCATE=H'00001000
            ;-------begin_of_function
                     .GLOBAL _func
            _func:
                     ST      RP, @-SP
                     ENTER   #4
            L_main:
                     LEAVE
                     LD      @SP+, RP
                     RET

Operation:  fcc935s  -s CODE=PROGRAM, CODE, 0x1000 -S -cpu MB93500 sample.c
Output:              .section  PROGRAM, CODE, LOCATE=H'00001000
                     .global   _func
            _func:
            L_20:
                     subi    sp, #16, sp
                     st      fp, @(sp, gr0)
                     mov     sp, fp
                     movsg   LR, gr4
                     sti     gr4, @(fp, 8)
            L_21:
            L_22:
                     ldi     @(fp, 8), gr4
                     movgs   gr4, LR
                     addi    fp, #16, sp
                     ld      @(fp, gr0), fp
                     ret
```

❍ **-splitsection option and -Xsplitsection option**

The -splitsection option generates CODE sections having different names for different functions.

This option can only be used with the fcc935s command.

The section naming conventions are as follows:

base-section-name_function-name_boundary-value

base-section-name is the section name assigned when neither the -splitsection nor the -align option is specified. The default section name is CODE. When the -s option or #pragma section is used to specify a section name, the section name is used as the base section name.

function-name is the name of the function defined in the file.

boundary-value is the boundary alignment value for the section. The default is 4. When the -align option is specified, its value is used as the boundary value.

If functions having the same name are contained in different files, sections are generated under the same name. Note that these sections are allocated in adjacent areas at the linking time.

The -splitsection option is ignored when the -s option or #pragma section is used to specify an arrangement address.

The -Xsplitsection option generates a single CODE section for each file. When both -

splitsection and -Xsplitsection are omitted, the -Xsplitsection option is used.

**[Remarks]**

The -splitsection option can be used to control the allocation order on a function-by-function basis at linking time.

With the FR500, an increase in the number of cache misses largely affects the performance. Two functions known to be frequently executed in succession may reduce instruction cache misses when they are allocated in contiguous areas in the order in which they are executed.

**[Output Example]**

```
Input:      int add(int a, int b) { return a+b; }
            int sub(int a, int b) { return a-b; }

Operation:  fcc935s -S -Kgs -cpu MB93500 sample.c -splitsection
Output:         .section  CODE_add_4, CODE, align=4    ;Section for function add
                .global   _add
_add:
                add.p     gr8, gr9, gr8
                                ret
                .section  CODE_sub_4, CODE, align=4    ;Section for function sub
                .global   _sub
_sub:
                sub.p     gr8, gr9, gr8
                                ret
```

❍ **-K {A1|A4|A8} option**

Specifies the minimum allocation boundary for external and static variables. The A4 suboption selects the 4-byte minimum allocation boundary. The A8 suboption selects the 8-byte minimum allocation boundary. The A8 suboption can only be used with the fcc935s command.

When the 4- or 8-byte minimum allocation boundary is used, more efficient code may be generated through inline expansion of a character string manipulation function when -K lib is specified. This code malfunctions if boundary alignment is incorrect. Therefore, objects for which different allocation boundaries are specified must not be linked. If such objects are linked, unnecessary areas are generated as a result of consistent boundary alignment, causing an increase in the number of objects. The A1 suboption selects the 1-byte minimum allocation boundary.

When this option is omitted, -K A1 is used.

**[Output Example]**

```
Input:      char c1, c2;
Operation:  fcc911s -K A4 -S -cpu MB91100 sample.c
            fcc935s -K A4 -S -cpu MB93500 sample.c
Output:             .SECTION    DATA, DATA, ALIGN=4
                    .GLOBAL _c2
            _c2:     .RES. B 4          ; Aligned on 4-byte bou
                    .GLOBAL _c1
            _c1:     .RES. B 4          ; Aligned on 4-byte bou
```

❍ **-K {SARG|DARG} option**

Specifies type of acquisition of area required for argument delivery to function.

This option can only be used in the fcc911s command.

When the DARG suboption is specified, dynamic allocation is achieved at function calling. This effectively decreases the stack consumption.

On the other hand, when the SARG suboption is specified, allocation is performed at the beginning of the caller function. In this case, the code size generally decreases with an increase in execution speed. However, stack use tends to increase.

If this option is not specified, -K SARG is applied.

**[Output Example]**

```
Input:     extern  void  sub(int, int, int, int, int);
           void  func(void) {sub(1, 2, 3, 4, 5);}
Operation: fcc911s -K  darg  -S  -cpu  MB91100  sample.c
Output:          LDI     #1, R4
                 LDI     #2, R5
                 LDI     #3, R6
                 LDI     #4, R7
                 LDI     #5, R0
                 ST      R0, @-SP      ; The argument area is allocated dynamically.
                 CALL32  _sub, R12
                 ADDSP   #4            ; The argument area is allocated dynamically.
```

❍ **-varorder {sort|normal} option**

The -varorder normal option causes static variables to be stored in memory in the order in which they are described in the source. Variables with an initial value and those without initial value are stored in different sections, so description with these sections mixed prevents the variables from being arranged in the order in which they are described in the source.

The -varorder sort option sorts the order of storing static variables in memory according to the order of variable alignment. Specifying this option reduces the size of unused static variable areas.

**[Example]**

```
Input Source:        long a;
                     char b;
                     long c;
                     char d;
Operation: fcc911s -S -varorder normal -cpu MB91100 sample.c
           fcc935s -S -varorder normal -cpu MB93500 sample.c
Output:
                     .ALIGN  4
                     .GLOBAL _a
           _a:
                     .RES.B  4
                     .ALIGN  1
                     .GLOBAL _b
           _b:
                     .RES.B  1
                     .ALIGN  4
                     .GLOBAL _c
           _c:
                     .RES.B  4
                     .ALIGN  1
                     .GLOBAL _d
           _d:
                     .RES.B  1
```

- **-t {none|used|all|local} option**

  This option is valid only for the C++ source. It specifies the type of template instantiation.

  The -t none option provides instantiation at object generation. Instantiation is done at linking.

  The -t used option instantiates only the parameter-type template functions and template member functions used in the module.

  The -t all option instantiates even unused member functions if they are of parameter type and template class used in the module.

  The -t local option generates template functions and template member functions used in the module as the in-module local functions.

  Note that overlapping instantiation by the used and all options is not allowed when more than one template is linked.

  For information on the template, see section 10.5 "Limitations on Use of C++ Template".

  The default is -t none.

- **--no_auto_instantiation option and --auto_instantiation option**

  These options are valid only for the C++ source.

  The --no_auto_instantiation option provides manual template instantiation. Instantiation must be controlled by the -t used option, -t all option, or #pragma.

  For information on using #pragma to control instantiation, see section 6.12 "Function for Controlling Installation of C++ Template."

- **--supress_vtbl option and --force_vtbl option**

  These options are valid only for the C++ source. They change the generation type of a virtual

function table.

The --supress_vtbl option causes the virtual function table to be referenced from other modules without being generated in a module.

The --force_vtbl option causes the virtual function table to be generated in a module for reference from other modules.

These options are valid only when the member functions specified as being virtual are in the class and all of them are defined inline. If these options are not given under this condition, a virtual function table is generated locally for each module

**[Example] Cases for option**

```
class FOO {
    int a;
public:
    FOO( ) {a=0;}
    virtual void memfunc(int x) {a=x;}
};  //  All the member functions are defined inline.
```

The --force_vtbl option must be specified for at least one module. Note that overlapping table definition is not allowed when specifying the option for more than one module.

## 3.5.7 Debug Information Related Options

**This section describes the options related to the debug information to be referenced by the symbolic debugger.**

### ■ Debug Information Related Options

#### ❍ -g option and -Xg option

Adds debug data to the object file.

When the optimization option (-O[1-4]) is specified, debugging should be performed noting the following points:. If the optimization option is specified, the compiler tries to assure better code output by changing the arithmetic operation target position and omitting any arithmetic operations that are judged to be unnecessary. To minimize the amount of data exchange with memory, the compiler tries to retain data within a register. It is therefore conceivable that a break point positioned in a certain line may fail to cause a break or that currently monitored certain address data may fail to vary with the expected timing. It also well to remember that the debug data will not be generated for an unused local variable or a local variable whose area need not be positioned in a stack as a result of optimization.

Debugging must be conducted with the above considerations taken into account.

The -Xg option cancels the -g option.

# 3.5.8 Command Related Options

**This section describes the options related to the other tools called by the command.**

## ■ Command Related Options

### ❍ -Y item, dir option and -XY option

Changes the item position to the dir directory. The -XY option cancels the -Y option. The item is one of the following.

- p: Left for compatibility with the previous version but does not provide anything.
- c: Changes the compiler pathname to dir.
- a: Changes the assembler pathname to dir.
- l: Changes the linker pathname to dir.

**[Example]**

```
>fcc911s file.c -Y c, \home\newlib -cpu MB91100
>fcc935s file.c -Y c, \home\newlib -cpu MB93500
```

Calls the compilar using \home\newlib\cpp as the path name.

### ❍ -T item, arg1 [, arg2...]option and -XT option

Passes arg to item as an individual compiler tool argument. The -XT option cancels the -T option.

Use a comma to separate arguments. To describe a comma as an argument, position a backslash (\) immediately before the comma. The comma positioned after the backslash will not be interpreted as a delimiter. To write a blank as an argument, describe a comma in place of a blank.

For the options for various commands, refer to the associated manuals. The following can be specified as the item.

- a: Assembler
- l: Linker

**[Example]**

```
>fcc911s -T a, -l, asmlist file.c -cpu MB91100
>fcc935s -T a, -l, asmlist file.c -cpu MB93500
```

Sequentially passes arguments "-l " and "asmlist" to the assembler. Therefore, the assemble list asmlist will be generated as a result of command execution.

# 3.5.9 Linkage Related Options

**The linkage related options are related to linkage.**

■ **Linkage Related Options**

❍ **-e name option and -Xe option**

The -e option sets the entry symbol to name at linking. The -Xe option cancels the -e option. Since the option definition is usually provided in the startup routine, this option does not have to be specified.

For details of the option, refer to the Linkage Kit Manual.

❍ **-L path option and -XL option**

The -L option adds path to the library path used at linking to search for a library to be linked. If the option is not specified, ${LIB911} is selected in the fcc911s command and ${LIB935} is selected in the fcc935s command automatically.

The -XL option cancels the -L option.

For details of the option, refer to the Linkage Kit Manual.

❍ **-l lib 1 [, lib 2...] option and -Xl option**

The -l option specifies the name (lib) of the library to be linked at linking.

If the extension entry is omitted, the .lib extension is added automatically.

The -Xl option cancels the -l option.

For the objects output by the compiler, by default, in fcc911s command lib911lib, in fcc935s command lib935.lib are set as the names of the libraries to be linked.

For the details of the option, refer to the Linkage Kit Manual.

❍ **-m option and -Xm option**

The -m option generates a map file at linking.

A map file output with a file name with the .map extension is generated in the current directory.

The -Xm option cancels the -m option.

❍ **-ra name = start/end option and -Xra option**

The -ra option specifies the RAM area at linking. The -Xra option cancels the -ra option. For details of the option, refer to the Linkage Kit Manual

❍ **-ro name = start/end option and -Xro option**

The -ro option specifies the ROM area at linking.The -Xro option cancels the -ro option.For details of the option, refer to the Linkage Kit Manual.

❍ **-sc param option and -Xsc option**

The -sc option specifies the section arrangement at linking.

The -Xsc option cancels the -sc option.

If the option is not specified, -sc IOPORT=0,*=0x1000 is selected automatically.

For details of the option, refer to the Linkage Kit Manual.

❍ **-startup filename option and -Xstartup option**

The -startup option selects filename as the object file name of the startup routine to be linked at linking.

If this -startup option is not specified, ${FETOOL}\lib\911\startup.obj is selected in the fcc911s command, and ${FETOOL}\lib\935\startup.obj is selected in the fcc935s command automatically.

The -Xstartup option cancels the -startup option.

For details of the startup routine, see CHAPTER 7 "EXECUTION ENVIRONMENT".

# 3.5.10 Option File Related Options

**This section describes the option file related options.**

## ■ Option File Related Options

### ❍ -f filename option and -Xf

-f filename option used to read the specified option file (filename). If the option file name does not have an extension, an .opt extension will be added.

The command options can be written in an option file. For the details of an option file, see section 3.6 "Option Files".

The -Xf option cancels the -f option.

### ❍ -Xdf option

-Xdf option specifies that the default option file will not be read.

For the default option file, see section 3.6 "Option Files".

# 3.6 Option Files

**This section explains option files for commands. By writing options in a file, a group of options can be specified. This feature also permits you to put startup options to be specified in a file.**

## ■ Option File

Option file reading takes place when an associated option is specified. This assures that the same result is obtained as when an option is specified at the -f specifying position in the command line.

If the option file name is without an extension, an .opt extension will be added.

### ❍ Option File General Format

All entries that can be made in a command line can be written in an option file.

A line feed in an option file is replaced by a blank.

A comment in an option file is replaced by a blank.

**[Example]**

```
-I  /usr/include     #  Include specifying
-D F2MC16            #  Macro specifying
-g                   #  Debug data generation specifying
-S                   #  Execution of processes up to compiling
```

## ■ Option File Limitations

The length of a line that can be written in an option file is limited to 4,095 characters.

The -f option can be written in an option file. However, nesting is limited to 8 levels.

## ■ Acceptable Comment Entry in Option File

A comment can be started from any column.

A comment is to begin with a sharp (#). The entire remaining portion of the line serves as the comment.

In addition, the following comments can also be used.

- /* Comment */
- * // Comment
- * ; Comment

**[Example]**

```
-I   /usr/include      # Include specifying
-D F2MC16              /* Macro specifying */
-g                     // Debug data generation specifying
-S                     ; Execution of processes up to compiling
```

## ■ Default Option File

A preselected option file can be read to initiate command execution. The obtained result will be the same as when an option is specified prior to another option specified in the command line.

The default option file name is predetermined as follows.

**[For UNIX OS]**

${OPT911}/fcc911.opt

${OPT935}/fcc935.opt

**[For Windows]**

%OPT911%\fcc911.opt

%OPT935%\fcc935.opt

The fcc911s command default option file name is fcc911.opt.

The fcc935s command default option file name is fcc935.opt.

If the default option file does not exist in the specified directory, such a specifying is ignored.

To inhibit the default option file feature, specify the -Xdof option in the command line.

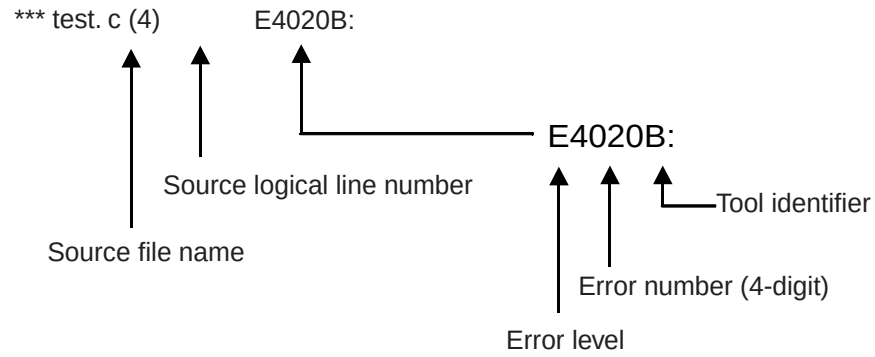
# 3.7 Messages Generated in Translation Process

**When an error is found in a source program or a condition which does not constitute a substantial error but requires attention is encountered, diagnostic messages may be generated at the time of translation. For message outputs generated by tools other than the compiler, refer to the respective manuals for the tool.**

## ■ Messages Generated in Translation Process

A diagnostic message output example is shown in Figure 3.7-1 "Diagnostic Message Example".

**Figure 3.7-1 Diagnostic Message Example**




## ■ Tool Identifier

The tool identifier indicates the tool that has detected the error.

- D: Driver
- B: Compiler
- S: Scheduler (Internal tool for compiler)
- A: Assembler
- L: Linker

## ■ Error Level

The error level represents the diagnostic check result type.

Table 3.7-1 "Relationship between Error Levels and Return Codes" describes the relationship between various error levels and return codes and their meanings.

**Table 3.7-1 Relationship between Error Levels and Return Codes**

| Error Level | Return Code | Meaning |
|---|---|---|
| I | 0 | Indicates a condition which does not constitute an error but requires attention. |
| W | 0 | Indicates a minor error. Process execution continues without being interrupted. The return code can be changed by the -cwno option. |

**Table 3.7-1 Relationship between Error Levels and Return Codes (Continued)**

| Error Level | Return Code | Meaning |
|---|---|---|
| E | 2 | Indicates a serious error. Process execution stops. |
| F | 3 | Indicates a fatal error which is related to quantitative limitations or system failure. Process execution stops. |

# CHAPTER 4 fcc911s COMMAND OBJECT PROGRAM STRUCTURE

**This chapter explains about the information necessary for program execution.**


4.1 "Section Structure of fcc911s"

4.2 "Rules for Name Generation with the fcc911s"

4.3 "fcc911s Command Boundary Alignment"

4.4 "fcc911s Command Bit Field"

4.5 "fcc911s Command Structure/Union"

4.6 "fcc911s Command Function Call Interface"

4.7 "fcc911s Command Interrupt Function Call Interface"

# 4.1 Section Structure of fcc911s

**The fcc911s command has the following seven sections:**

- **Code section**
- **Initialized section**
- **Constant section**
- **Data section**
- **I/O section**
- **Vector section**
- **C++ Init section**

### ■ fcc911s Command Section Structure

Table 4.1-1 "fcc911s Command Section List" shows the sections to be generated by the compiler and their meanings.

**Table 4.1-1 fcc911s Command Section List**

| No. | Section Type | Section Name | Type | Boundary Alignment [Byte] | Write | Initial Value |
|---|---|---|---|---|---|---|
| 1 | Code section | CODE | CODE | 2 | Disabled | Provided |
| 2 | Initialized section | INIT | DATA | 4 | Enabled | Provided |
| 3 | Constant section | CONST | CONST | 4 | Disabled | Provided |
| 4 | Data section | DATA | DATA | 4 | Enabled | Not provided |
| 5 | I/O section | IO | IO | 4 | Enabled | Not provided |
| 6 | Vector section | INTVECT | DATA | 4 | Disabled | Provided |
| 7 | C++Init section | EXT_CTOR_ DTOR | CONST | 4 | Disabled | Provided |

The purpose of each section use and the relationship to the C/C++ language are explained below.

❍ **Code section**

Code section stores machine codes. This section corresponds to the procedure section for the C language.

❍ **Initialized section**

Initialized section stores the initial value attached variable area. For the C language, this section corresponds to the area for external variables without the const attribute, static external variables, and static internal variables.

❍ **Constant section**

Constant section stores the write-protected initial value attached variable area. For the C

language, this section corresponds to the area for const attribute attached external variables, static external variables, and static internal variables.

- **Data section**

  Data sectio stores the area for variables without the initial value. For the C language, this section corresponds to the area for external variables (including those which are with the const attribute), static external variables, and static internal variables.

- **I/O section**

  I/O section stores the area for the __io-qualified variables. For the C language, this section corresponds to the area for __io-qualified external variables (including those which are provided with the const attribute), static external variables, and static internal variables.

  The default section name is IO.

- **Vector section**

  This section is where interrupt vector tables are stored. In C, a vector table is generated only when its generation is specified in #pragma intvect. The default section name is INTVECT.

- **C++ Init section**

  This section is where tables for indicating the entry of functions constituting and destroying static objects are stored. It must be used at start-up. For information on specifying the startup program, see section 7.2 "Startup Routine Creation".

# 4.2 Rules for Name Generation with the fcc911s

**The rules for the names used by the compiler are explained below.**

## ■ Rules for Name Generation with the fcc911s

Table 4.2-1 "Label Generation Rules" shows the relationship between the names generated by the compiler and the C language.

**Table 4.2-1 Label Generation Rules**

| C Language Counterpart | Label Generated by Compiler |
|---|---|
| Function name | -function name |
| External variable name | -external variable name |
| Static variable name | LI_no |
| Local variable name | - |
| Virtual argument name | - |
| Character string, derived type | LS_no |
| Automatic variable initial value | LS_no |
| Target location label | L_no |

Note: The compiler internal generation number is placed at the no position.

# 4.3 fcc911s Command Boundary Alignment

**The standard data type and boundary alignment is explained below. Table 4.3-1 "fcc911s Command Variable Assignment Rules" shows the assignment rules.**

## ■ fcc911s Command Boundary Alignment

**Table 4.3-1 fcc911s Command Variable Assignment Rules**

| Variable Type | Assignment Size [Byte] | Boundary Alignment [Byte] |
|---|---|---|
| char | 1 | 1 |
| signed char | 1 | 1 |
| unsigned char | 1 | 1 |
| short | 2 | 2 |
| unsined short | 2 | 2 |
| int | 4 | 4 |
| unsigned int | 4 | 4 |
| long | 4 | 4 |
| unsigned long | 4 | 4 |
| float | 4 | 4 |
| double | 8 | 4 |
| long double | 8 | 4 |
| pointer/address | 4 | 4 |
| Structure/union/class | Explained later | Explained later |

Note: Some variables are aligned on 4-byte boundaries when the -K A4 option is specified. The -K A4 option does not affect structure/union member boundary alignment.

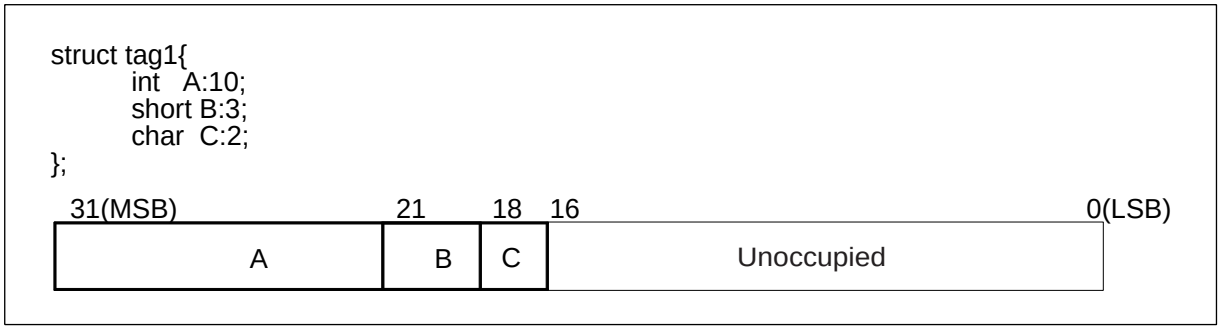
# 4.4 fcc911s Command Bit Field

**The bit field data size and boundary alignment is explained below. The bit field data is assigned to a storage unit that has an adequate size for bit field data retention and is located at the smallest address.**

## ■ fcc911s Command Bit Field

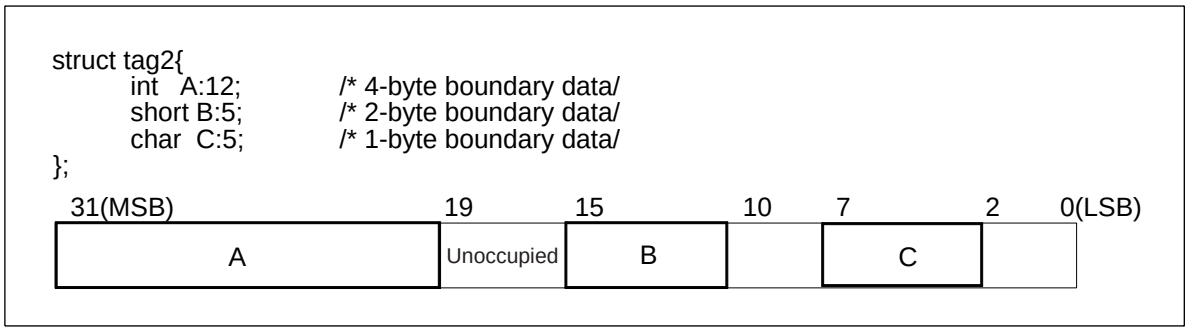
Consecutive bit field data are packed at consecutive bits having the same storage unit, without regard to the type, beginning with the LSB and continuing toward the MSB. An example is shown in Figure 4.4-1 "fcc911s Command Bit Field Data Size and Boundary Alignment Example 1".

**Figure 4.4-1 fcc911s Command Bit Field Data Size and Boundary Alignment Example 1**



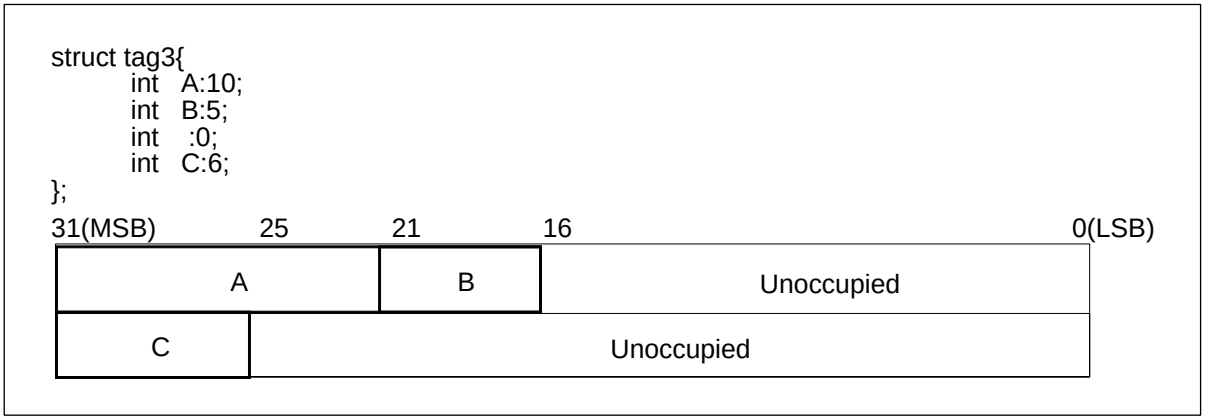

If a field to be assigned lies over a bit field type boundary, its assignment is completed by aligning it with a boundary suitable for the type. An example is shown in Figure 4.4-2 "fcc911s Command Bit Field Data Size and Boundary Alignment Example 2".

**Figure 4.4-2 fcc911s Command Bit Field Data Size and Boundary Alignment Example 2**




When a bit field having a bit length of 0 is declared, it is forcibly assigned to the next storage unit. An example is shown in Figure 4.4-3 "fcc911s Command Bit Field Data Size and Boundary Alignment Example"

**Figure 4.4-3 fcc911s Command Bit Field Data Size and Boundary Alignment Example**



# 4.5 fcc911s Command Structure/Union

**The structure/union of fcc911s command data size and boundary alignment are explained below. The structure/union data size is a multiple of the maximum boundary alignment size of the members. Boundary alignment for the area itself is accomplished by means of member maximum boundary alignment. The individual members are subjected to boundary alignment in accordance with the member type.**

## ■ fcc911s Command Structure/Union

Figure 4.5-1 "fcc911 Command Structure/Union Data Size and Boundary Alignment Example 1" to Figure 4.5-3 "fcc911 Command Structure/Union Data Size and Boundary Alignment Example 3" show examples concerning structure/union data size and boundary alignment.

**Figure 4.5-1 fcc911 Command Structure/Union Data Size and Boundary Alignment Example 1**

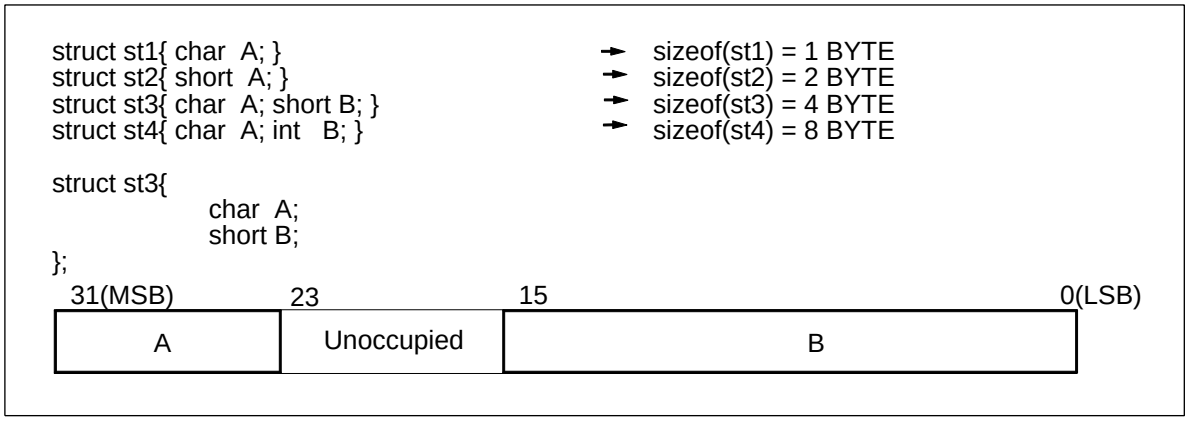



**Figure 4.5-2 fcc911 Command Structure/Union Data Size and Boundary Alignment Example 2**

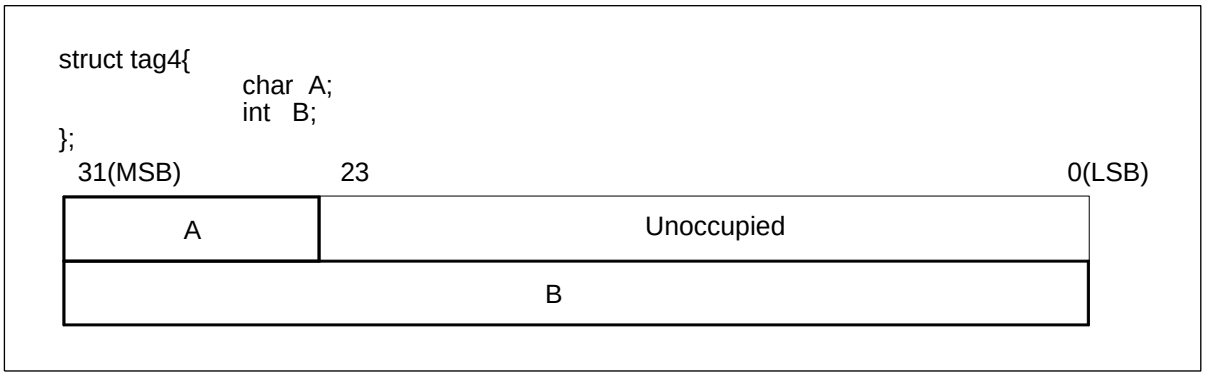
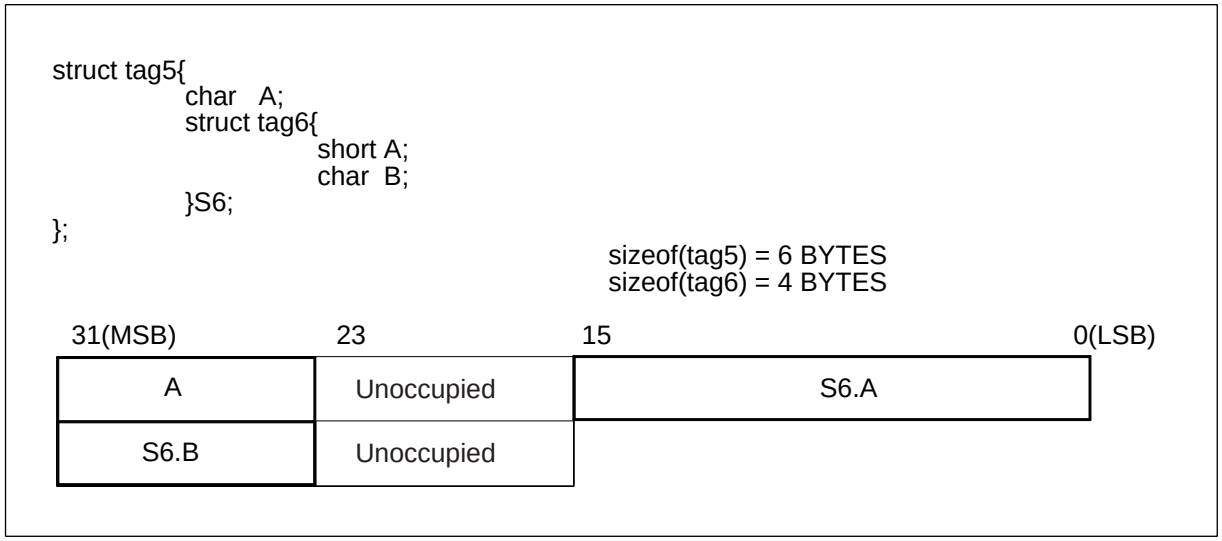


**Figure 4.5-3 fcc911 Command Structure/Union Data Size and Boundary Alignment Example 3**



# 4.6 fcc911s Command Function Call Interface

**The general rules for control transfer between functions are established as standard regulations for individual architectures and are called standard linkage regulations. A module written in C language can be combined with a module written using a different method (e.g., assembler language) when the standard linkage regulations are complied with.**

## ■ fcc911s Command Function Call Interface

- **Stack frame**

  The stack frame construction is stipulated by the standard linkage regulations.

- **Argument**

  Argument transfer relative to the callee function is effected via a stack or register.

- **Argument extension format**

  When an argument is to be stored in a stack, the argument type is converted to an extended format in accordance with the argument type.

- **Calling procedure**

  The caller function initiates branching to the callee function after argument storage.

- **Register**

  The register guarantee stated in the standard linkage regulations and the register setup regulations are explained later.

- **Return value**

  The return value interface stated in the standard linkage regulations is explained later.

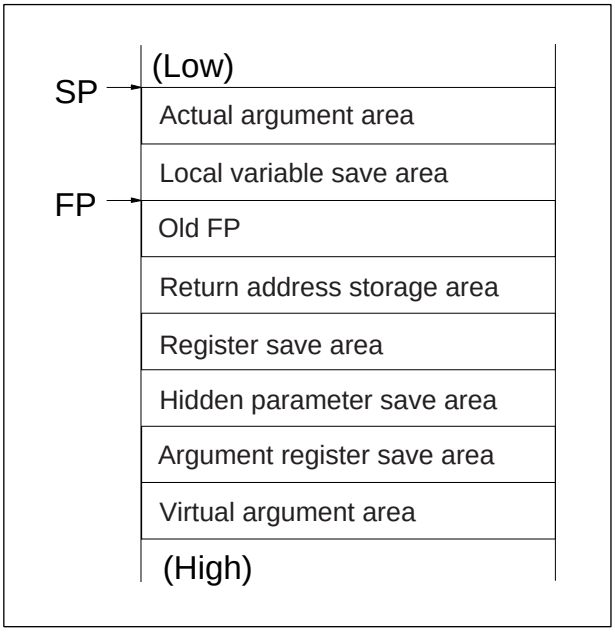
# 4.6.1 fcc911s Command Stack Frame

**The standard linkage regulations prescribe the stack frame construction.**

## ■ fcc911s Command Stack Frame

The stack pointer (SP) always indicates the lowest order of the stack frame. Its address value always represents the work boundary. Figure 4.6-1 "fcc911s Command Stack Frame" shows the standard function stack frame status.

**Figure 4.6-1 fcc911s Command Stack Frame**




### ❍ Actual argument area/virtual argument area

When a function is called, this area is used for argument transfer. This area is referred to an actual argument area when the caller function is used to set up an argument and to a virtual argument area when the argument is referenced by the caller function. This area is created when all arguments cannot be contained in an argument register during the transfer of an argument.

For details, see section 4.6.2 "fcc911s Command Argument".

### ❍ Local variable area

This is the area for local variables and temporary variables.

### ❍ Old FP

This area stores the frame pointer (FP) value of the caller function.

### ❍ Return address storage area

This area saves the RP. The RP stores the address of a return to the caller function for the purpose of function calling.

- **Register save area**

  This is a register save area that must be guaranteed for the caller function. This area is not secured when the register save operation is not needed.

- **Hidden parameter save area**

  This area stores the start address of the return value storage area for a structure/union return function.

  When a structure/union is used as the return value, the caller function stores the return value storage area start address in register R4 and passes it to the caller function.

  The callee function interprets the address stored in the R4 as the return value storage area start address.

  When register R4 needs to be saved into memory, the callee function saves it in the hidden parameter save area. This area is not secured when the save operation is not needed.

- **Argument register save area**

  This area saves the argument register. This area is not secured when the save operation is not needed.

  For details, see section 4.6.2 "fcc911s Command Argument".

## 4.6.2 fcc911s Command Argument

**Arguments, the count of which equals the count of argument registers (4 words), are positioned in registers R4 to R7 and delivered to the callee function. When a structure/union return function is called, three argument registers (R5 to R7) are used because the return value area address is stored in register R4. Arguments not placed in the argument registers will be stored in the stack actual argument area for transfer purposes. When an 8-byte type argument is to be delivered using registers, it is divided into two and placed in two registers for transfer.**

### ■ fcc911s Command Argument

When argument registers must be saved to memory, the callee function secures an argument register save area in the stack. In this case, a continuous argument register save area must be established in the virtual argument area. The argument register save area must be allocated as needed to cover the size of the argument register to be saved.

If the function has a variable count of arguments, it saves all argument registers in the argument register save area.

If the function has a variable count of arguments, it saves all argument registers in the argument register save area.

**<Caution>**

In a C++ program, arguments that do not appear in the source program may be passed. The order and location in which arguments are stacked may or may not be as desired.

**[Example 1]**

```
double d;
sub(d);   → The high-order words of d are delivered by R4,
            and the low-order words of d are delivered by R5.
```

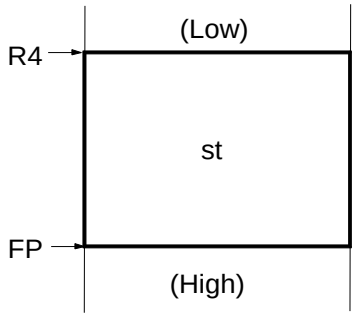
**[Example 2]**

```
int  a, b, c;
double d;
sub(a, b, c, d); → a is delivered by R4, b by R5, and c by R6.
                   The high-order words of d are delivered by R7,
                   and the low-order words of d are delivered by
                   the stack.
```

When a structure/union is to be delivered as an argument, the caller copies the structure to the local variable area and passes the address of that area to the callee. In this case, if the structure/union size is less than 4 bytes or is not divisible by 4, the less-than-4-byte fraction is handled as one 4-byte unit.

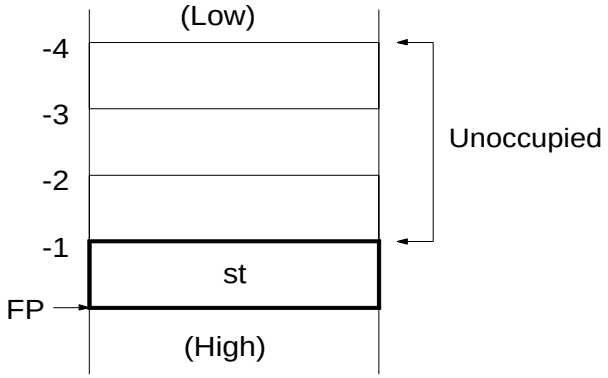
**[Example 3]**

```
struct A st;
sub(st);
```




**[Example 4]**

```
struct A{ char a; }st;
```

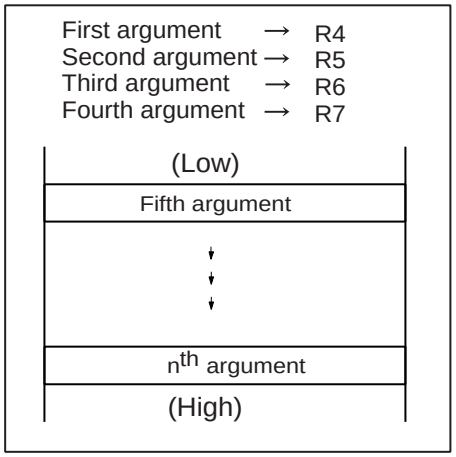



When a function receiving a variable count of arguments is to be called, the arguments are placed in registers in the same manner as for transfer. The called function stores all the register-delivered arguments in the argument register save area in the stack.

The actual argument area is allocated/deallocated by the caller function, whereas the argument register save area is allocated/deallocated by the callee function.

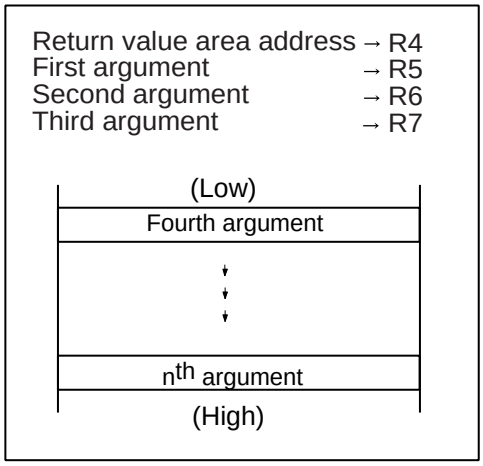
Figure 4.6-2 "fcc911s Command Argument Format Stated in Standard Linkage Regulations" and Figure 4.6-3 "Argument Format for fcc911s Command Structure/Union Return Function Calling" show the argument formats prescribed in the standard linkage regulations.

**Figure 4.6-2 fcc911s Command Argument Format Stated in Standard Linkage Regulations**




Note: Two argument registers are required for 8-byte type arguments.

**Figure 4.6-3 Argument Format for fcc911s Command Structure/Union Return Function Calling**




Note: Two argument registers are required for 8-byte type arguments.

## 4.6.3 fcc911s Command Argument Extension Format

**When an argument is to be stored in the stack, its type is converted to an extended type in accordance with the individual argument type. The argument is freed by the caller function after the return from the callee function is made.**

### ■ fcc911s Command Argument Extension Format

Table 4.6-1 "fcc911 Command Argument Extension Format" shows the argument extension format.

**Table 4.6-1 fcc911 Command Argument Extension Format**

| Actual Argument Type | Extended Type *1 | Boundary Strage Size [Byte] |
|---|---|---|
| char | int | 4 |
| signed char | int | 4 |
| unsigned char | int | 4 |
| short | int | 4 |
| unsined short | int | 4 |
| int | No extension | 4 |
| unsigned int | No extension | 4 |
| long | No extension | 4 |
| unsigned long | No extension | 4 |
| float | double | 8 |
| double | No extension | 8 |
| long double | No extension | 8 |
| pointer/address | No extension | 4 |
| Structure/union | - | 4 *2 |
| Class | - | 4 *3 |

*1: The extended type represents an extended type that is provided when no argument type is given. When a prototype declaration is made, it is complied with.

*2: When a structure/union is to be delivered as an argument, the caller copies it to the local variable area and delivers the address of that area.

*3: Passing of a class as an argument depends on the availability and contents of a copy constructor. Without the copy constructor, the class is passed in the same manner as the structure.

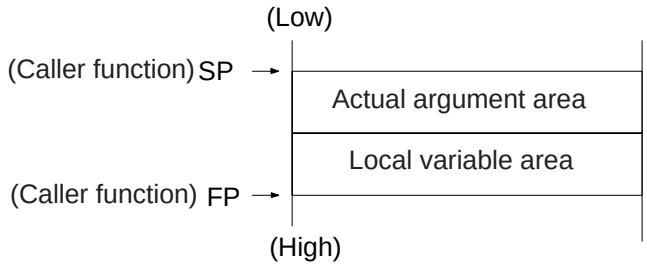
# 4.6.4 fcc911s Command Calling Procedure

**The caller function initiates branching to the callee function after argument storage.**

## ■ fcc911s Command Calling Procedure

Figure 4.6-4 "fcc911s Stack Frame Prevailing at Calling in Compliance with fcc911s Command Standard Linkage Regulations" shows the stack frame prevailing at calling in compliance with the standard linkage regulations.

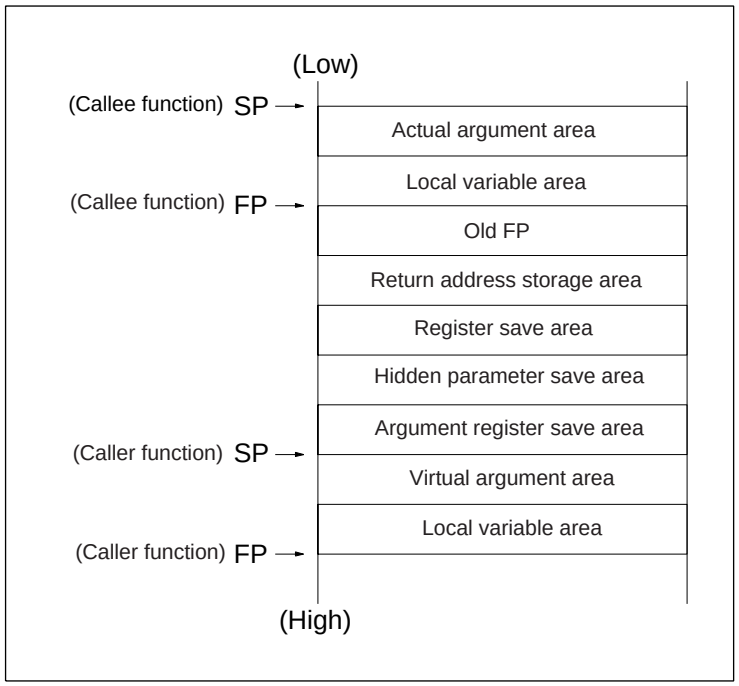
**Figure 4.6-4 fcc911s Stack Frame Prevailing at Calling in Compliance with fcc911s Command Standard Linkage Regulations**




The callee function saves the caller function frame pointer (FP) in the stack and then stores the prevailing stack pointer value in the stack as the new frame pointer value. Subsequently, the local variable area and caller function register save area are acquired from the stack to save the caller register.

Figure 4.6-5 "Stack Frame Created by Callee Function in Compliance with fcc911s Command Standard Linkage Regulations" shows the stack frame that is created by the callee function in compliance with the standard linkage regulations

**Figure 4.6-5 Stack Frame Created by Callee Function in Compliance with fcc911s Command Standard Linkage Regulations**



## 4.6.5 fcc911s Command Register

**This section states the register guarantee and register setup regulations in the standard linkage regulations.**

### ■ fcc911s Command Register Guarantee

The callee function guarantees the following registers of the caller function.

- General-purpose registers R8 to R11, R14, and R15

The register guarantee is provided when the callee function acquires a new area from the stack and saves the register value in that area. Note, however, that registers remaining unchanged within the function are not saved. If such registers are altered using the ASM statement, etc., no subsequent operations will be guaranteed.

### ■ fcc911s Command Register Setup

The register regulations for function call and return periods are indicated in Table 4.6-2 "Register Regulations for fcc911s Command Function Call and Return Periods".

**Table 4.6-2 Register Regulations for fcc911s Command Function Call and Return Periods**

| Register | Call Period | Return period |
|---|---|---|
| R4 | Argument/return value area address*1 | Return value*2 |
| R5 | Argument register*1 | Return value*3 |
| R6 and R7 | Argument register*1 | Not stipulated |
| R0 to R3 | Not stipulated | Not stipulated |
| R12 and R13 | Not stipulated | Not stipulated |
| R8 to R11 | Not stipulated | Call period value guaranteed |
| R14 | Frame pointer (FP) | Call period value guaranteed |
| R15 | Stack pointer (SP) | Call period value guaranteed |

*1: There are no stipulations for unused registers in situations where the argument is less than 4 words.

*2: There are no stipulations for situations where a function without the return value is called or a function with a structure/union type return value is called.

*3: There are no stipulations for situations where the function to be called has a return value other than a double or long double type.

## 4.6.6 fcc911s Command Return Value

**The return value interface stated in the standard linkage regulations is indicated in Table 4.6-3 "fcc911 Command Return Value Interface Stated in Standard Linkage Regulations".**

■ **fcc911s Command Return Value**

**Table 4.6-3 fcc911 Command Return Value Interface Stated in Standard Linkage Regulations**

| Return Value Type | Return Value Interface |
|---|---|
| void | None |
| char | R4 |
| signed char | R4 |
| unsigned char | R4 |
| short | R4 |
| unsined short | R4 |
| int | R4 |
| unsigned int | R4 |
| long | R4 |
| unsigned long | R4 |
| float | R4 |
| double | R4 and R5 *1 |
| long double | R4 and R5 *1 |
| Pointer/address | R4 |
| Structure/union | R4 *2 |
| Class | R4 *3 |

*1: The 4 high-order bytes of a total of 8 bytes are stored in R4 and the remaining 4 low-order bytes are stored in R5.

*2: When a structure/union is used as the return value, the caller function stores the start address of the return value storage area into R4 and then passes it to the callee function. The callee function interprets R4 as the start address of the return value storage area. When this address needs to be saved in memory, the callee function secures the hidden parameter save area and saves the address in that area.

*3: Passing of a class as a return value depends on the availability and contents of a copy constructor. Without the copy constructor, the class is passed in the same manner as the structure.

# 4.7 fcc911s Command Interrupt Function Call Interface

**The interrupt function can be written using the __interrupt type qualifier. If the interrupt function is called by a method other than an interrupt, no subsequent operations will be guaranteed. The function call interface within the interrupt function is the same as stated in the standard linkage regulations.**

## ■ fcc911s Command Interrupt Function Call Interface

❍ **Interrupt stack frame**

When an interrupt occurs, the stack is changed to the interrupt stack.

❍ **Argument**

No argument can be specified for the interrupt function. If any argument is specified for the interrupt function, no subsequent operations will be guaranteed.

❍ **Interrupt function calling procedure**

The interrupt function is called by an interrupt via the interrupt vector table. If the interrupt function is called by any other method, no subsequent operations will be guaranteed.

❍ **Register**

As regards the interrupt function, all registers are guaranteed.

❍ **The return value**

Interrupt function does not usually have a return value.

# 4.7.1 fcc911s Command Interrupt Stack Frame

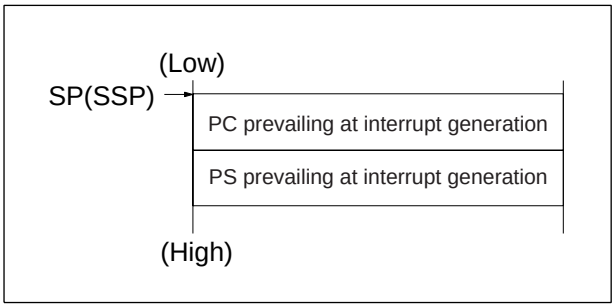
**When an interrupt occurs, the stack is changed to the interrupt stack.**

■ **Interrupt Stack Frame**

When an interrupt occurs, the stack pointer (SP) is replaced by the interrupt stack pointer (SSP). Within the interrupt function, the interrupt stack pointer is used as the normal stack pointer.

Figure 4.7-1 "fcc911 Command Interrupt Stack Frame" shows the interrupt stack frame status prevailing immediately after interrupt generation.

**Figure 4.7-1 fcc911 Command Interrupt Stack Frame**



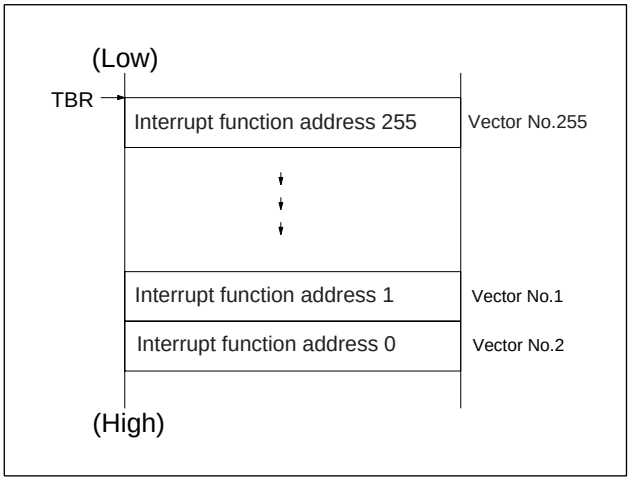
## 4.7.2 fcc911s Command Interrupt Function Calling Procedure

**The interrupt function is called by an interrupt via the interrupt vector table. If the interrupt function is called by any other method, no subsequent operations will be guaranteed.**

### ■ Interrupt Function Calling Procedure

Figure 4.7-2 "fcc911 Command Interrupt Vector Table" shows an example interrupt vector table.

**Figure 4.7-2 fcc911 Command Interrupt Vector Table**




When an interrupt is generated, the vector table corresponding to the interrupt vector number is referenced according to the following calculation.

TBR + O × 3FC - (4 * vector number)

For the details of interrupts, refer to the FR20 Architecture Manual.

# CHAPTER 5 fcc935s COMMAND OBJECT PROGRAM STRUCTURE

**This chapter explains the requirements for program execution.**


5.1 "Section Structure of fcc935s"

5.2 "Generation Rules for Names Used by fcc935s"

5.3 "fcc935s Command Boundary Alignment"

5.4 "fcc935s Command Bit Field"

5.5 "fcc935s Command Structure/Union"

5.6 "fcc935s Command Function Call Interface"

# 5.1 Section Structure of fcc935s

**The fcc935s command has the following six sections:**

- **Code section**
- **Initialized section**
- **Constant section**
- **Data section**
- **I/O section**
- **C++ Init section**

## ■ fcc935s Command Section Structure

Table 5.1-1 "fcc935s Command Section List" lists the sections to be generated by the compiler and their meanings.

**Table 5.1-1 fcc935s Command Section List**

| No. | Section type | Section name | Type | Boundary alignment (in bytes) | Write | Initial value |
|---|---|---|---|---|---|---|
| 1 | Cide section | CODE | CODE | 4 | Disabled | Provided |
| 2 | Initialized section | INIT | DATA | 8 | Enabled | Provided |
| 3 | Constant section | CONST | CONST | 8 | Disabled | Provided |
| 4 | Data section | DATA | DATA | 8 | Enabled | Not provided |
| 5 | I/O section | IO | IO | 8 | Enabled | Not provided |
| 6 | C++ Init section | EXT_CTOR_DTOR | CONST | 4 | Disabled | Provided |

The usage purpose of each section and the relationship of the section to C/C++ are explained below.

### ❍ Code section

The code section stores machine codes. This section corresponds to the procedure section for C.

### ❍ Initialized section

The initialized section stores the area for variables with initial values. For C, this section corresponds to the area for external variables without the const attribute, static external variables, and static internal variables.

### ❍ Constant section

The constant section stores the area for write-protected variables with initial values. For C, this section corresponds to the area for external variables with const attribute, static external variables, and static internal variables.

- **Data section**

  The data section stores the area for variables without initial value. For C, this section corresponds to the area for external variables (including variables with const attribute), static external variables, and static internal variables.

- **I/O section**

  The I/O section stores the area for variables with the _ _io-type modifier. For C, this section corresponds to the area for external variables that have the _ _io type modifier (including variables with the const attribute), static external variables, and static internal variables. The default section name is IO.

- **C++ Init section**

  The C++ Init section stores the table that points to the entry of functions creating and deleting static class objects. This section must be used at startup. For information on specifying the startup program, see Section 7.2 "Startup Routine Creation".

# 5.2 Generation Rules for Names Used by fcc935s

**The rules for the names used by fcc935s are explained below.**

## ■ Generation Rules for Names Used by fcc935s

Table 5.2-1 "Label Generation Rules" shows the relationship between the names generated by the compiler and C.

**Table 5.2-1 Label Generation Rules**

| Corresponding item in C | Label generated by compiler |
|---|---|
| Function name | _function name |
| External variable name | _external variable name |
| Static external variable name | _static external variable name |
| Static variable name | LI_no |
| Local variable name | — |
| Virtual argument name | — |
| Character string, derived type | LS_no |
| Automatic variable initial value | LS_no |
| Target location label | L_no |

Note: The compiler internal generation number is inserted in place of "no".

# 5.3 fcc935s Command Boundary Alignment

**The standard data types and boundary alignment are explained below. Table 5.3-1 "fcc935s Command Variable Assignment Rules" shows the assignment rules.**

## ■ fcc935s Command Boundary Alignment

**Table 5.3-1 fcc935s Command Variable Assignment Rules**

| Variable type | Assignment size (byte) | Boundary alignment (byte) |
|---|---|---|
| char | 1 | 1 |
| signed char | 1 | 1 |
| unsigned char | 1 | 1 |
| short | 2 | 2 |
| unsigned short | 2 | 2 |
| int | 4 | 4 |
| unsigned int | 4 | 4 |
| long | 4 | 4 |
| unsigned long | 4 | 4 |
| long long | 8 | 8 |
| unsigned long long | 8 | 8 |
| float | 4 | 4 |
| double | 8 | 8 |
| long double | 8 | 8 |
| Pointer/address | 4 | 4 |
| Structure/union | Explained later | Explained later |

Note: Some variables are aligned on 4-byte boundaries when the -K A4 option is specified, or aligned on 8-byte boundaries when the -K A8 option is specified. The -K A4 and -K A8 option do not affect structure/union member boundary alignment.

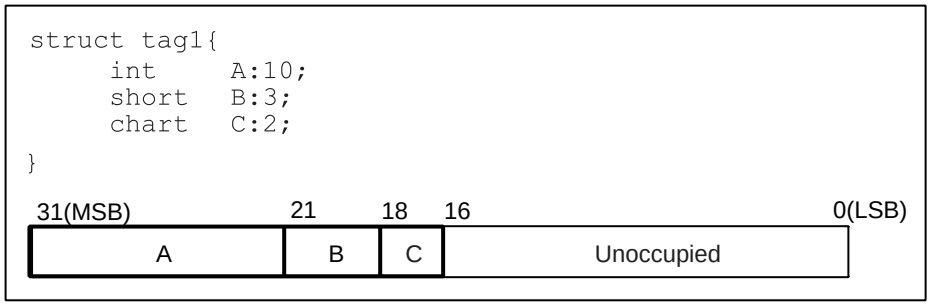
# 5.4 fcc935s Command Bit Field

**The fcc935s command bit field data size and boundary alignment are explained below. The bit field data is assigned to a storage unit that has an adequate size for bit field data retention and is located at the smallest address.**

## ■ fcc935s Command Bit Field

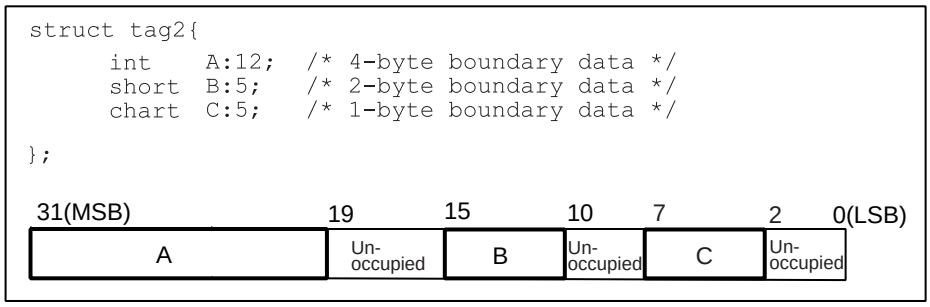
Consecutive bit field data is packed at consecutive bits having the same storage unit, beginning with LSB and continuing toward the MSB regardless of the type. An example is shown in Figure 5.4-1 "fcc935s Command Bit Field Data Size and Boundary Alignment Example".

**Figure 5.4-1 fcc935s Command Bit Field Data Size and Boundary Alignment Example 1**




If a field for assignment lies beyond a bit field type boundary, its assignment is completed by aligning it with a boundary suitable for the field type. An example is shown in Figure 5.4-2 "fcc935s Command Bit Field Data Size and Boundary Alignment Example".

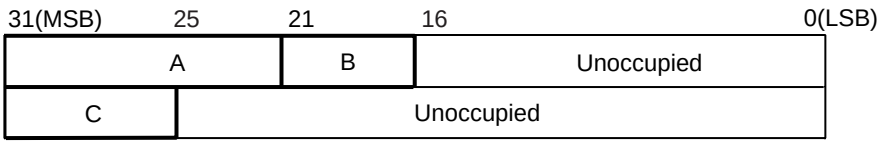
**Figure 5.4-2 fcc935s Command Bit Field Data Size and Boundary Alignment Example 2**




When a bit field having a bit length of 0 is declared, it is forcibly assigned to the next storage unit. An example is shown in Figure 5.4-3 "fcc935s Command Bit Field Data Size and Boundary Alignment Example 3".

**Figure 5.4-3 fcc935s Command Bit Field Data Size and Boundary Alignment Example 3**

```
struct tag3{
    int   A:10;
    int   B:5;
    int    :0;
    int   C:6;
};
```



# 5.5 fcc935s Command Structure/Union

**The fcc935s command structure/union data size and boundary alignment are explained below. The structure/union data size is a multiple of the maximum boundary alignment size of the members. Boundary alignment for the area itself is accomplished by means of member maximum boundary alignment. The individual members are subjected to boundary alignment in accordance with the member type.**

## ■ fcc935s Command Structure/Union

Figure 5.5-1 "fcc935s Command Structure/union Data Size and Boundary Alignment Example 1", Figure 5.5-2 "fcc935s Command Structure/union Data Size and Boundary Alignment Example 2", and Figure 5.5-3 "fcc935s Command Structure/union Data Size and Boundary Alignment Example 3" show examples concerning structure/union data size and boundary alignment.

**Figure 5.5-1 fcc935s Command Structure/union Data Size and Boundary Alignment Example 1**

```
 struct st1{ char A; }           → sizeof(st1) = 1 BYTE
 struct st2{ short A; }          → sizeof(st2) = 2 BYTE
 struct st3{ char A; short B; }  → sizeof(st3) = 4 BYTE
 struct st4{ char A; int B; }    → sizeof(st4) = 8 BYTE

 struct st3{
             char  A;
             short B;
};
```

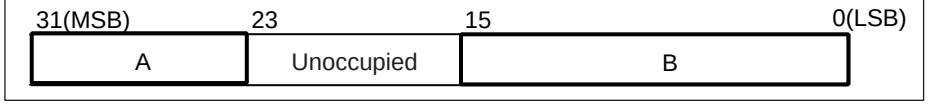



**Figure 5.5-2 fcc935s Command Structure/union Data Size and Boundary Alignment Example 2**

```
struct tag4{
           chart   A;
           int     B;

};
```

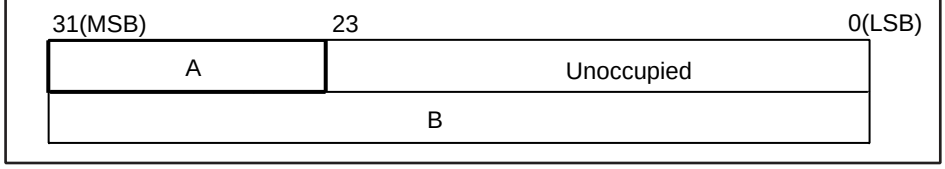
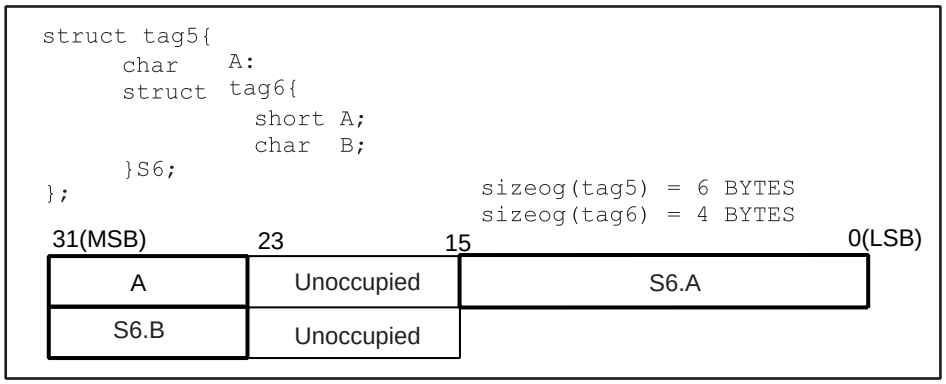


**Figure 5.5-3 fcc935s Command Structure/union Data Size and Boundary Alignment Example 3**



# 5.6 fcc935s Command Function Call Interface

**The general rules for control transfer between functions are established as standard regulations for individual architectures and are called standard linkage regulations. A module written in C can be combined with a module coded using a different method (e.g., assembler language) when the standard linkage regulations are complied with.**

## ■ fcc935s Command Function Call Interface

❍ **Stack frame**

The stack frame construction is stipulated by the standard linkage regulations.

❍ **Argument**

Argument transfer relative to the called function is effective via a stack or register.

❍ **Argument extension format**

When an argument is to be stored in a stack, the argument type is converted to an extended format in accordance with the argument type.

❍ **Calling procedure**

The caller function initiates branching to the called function after argument storage.

❍ **Register**

The register assurance stated in the standard linkage regulations and the register setup regulations are explained later.

❍ **Return value**

The return value interface stated in the standard linkage regulations is explained later.

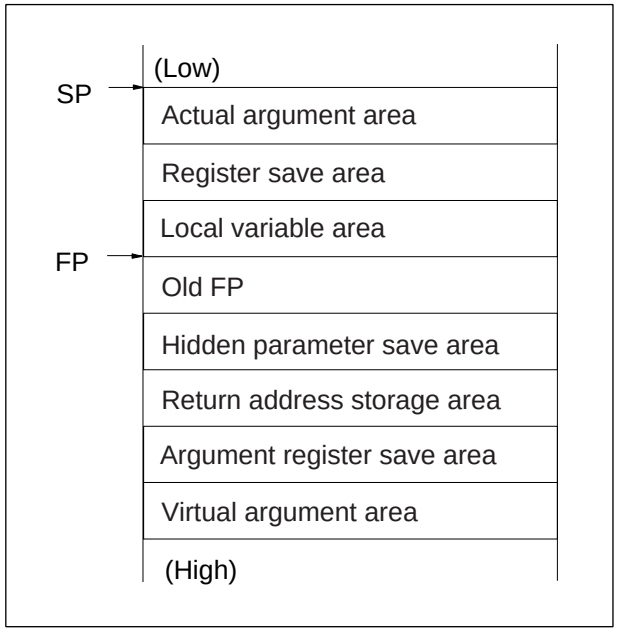
# 5.6.1 fcc935s Command Stack Frame

**The standard linkage regulations stipulate the stack frame construction.**

## ■ fcc935s Command Stack Frame

The stack pointer (SP) always points to the lowest order of the stack frame. Its address value always represents the 8-byte boundary. Figure 5.6-1 "fcc935s Command Stack Frame" shows the standard function stack frame status. The following area size is rounded up in 8 byte increments due to 8-byte boundary alignment.

**Figure 5.6-1 fcc935s Command Stack Frame**




The following area size is rounded up each 8 bytes due to 8-byte boundary alignment:

- Actual argument area/virtual argument area
- Register save area
- Local variable area
- Argument register save area

### ❍ Actual argument area/virtual argument area

When a function is called, this area is used for argument transfer. This area is referred to as an actual argument area when the caller function is used to set an argument, and referred to as a virtual argument area when the argument is referenced via the called function. This area is created when not all arguments cannot be stored in an argument register during argument transfer. For details, see Section 5.6.2 Fcc935s Command Argument".

### ❍ Local variable area

This area is used for local variables and temporary variables.

- **Old FP**

  This area stores the frame pointer (FP) value of the caller function.

- **Return address storage area**

  This area saves the LR. The LR stores the address for a return to the caller function for the purpose of function calling.

- **Register save area**

  This area saves the register that must be guaranteed for the caller function. When the register need not be saved, this area is not allocated.

- **Hidden parameter save area**

  This area stores the start address of the return value storage area for a structure/union return function. When a structure/union is used as the return value, the caller function stores the start address of the return value storage area in register GR3 and passes it to the called function. The called function interprets the address stored in GR3 as the start address of the return value storage area. When the register GR3 number be saved into memory, the called function saves it in the hidden parameter save area.

- **Argument register save area**

  This area saves the argument register. This area is not allocated when the save operation is not needed. For details, see Section 5.6.2 "Fcc935s Command Argument".

## 5.6.2 fcc935s Command Arguments

**Arguments are placed in registers GR8 to GR13 and passed to the called function for the number of argument registers (6 words). Arguments not placed in the argument registers will be stored in the stack actual argument area for transfer purposes. When an 8-byte argument is to be passed using registers, it is divided into two and placed in two registers for transfer.**

### ■ fcc935s Command Arguments

When argument registers must be saved to memory, the called function allocates the argument register save area in the stack. In this case, a continuous argument register save area must be established in the virtual argument area. The function that has a variable count of arguments saves all of the argument registers in the argument register save area.

**[Example 1]**

```
double d;
sub(d);   → The high-order words of d are delivered by GR8,
            and the low-order words of d are delivered by GR9.
```

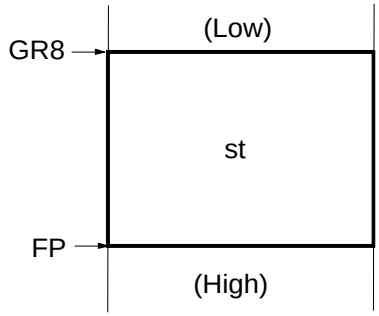
**[Example 2]**

```
int  a, b, c, d, e;
double f;
sub(a, b, c, d, e, f);   → a, b, c, d, and e are delivered by GR8, GR9, GR10, GR11,
                           and GR12. The high-order words of f are delivered by GR13,
                           and the low-order words of f are delivered by the stack.
```

When a structure/union is to be delivered as an argument, the caller copies the structure to the local variable area and passes the address of that area to the callee. In this case, the structure/union size that is less than 4 bytes or cannot be divided by 4 is rounded up in 4-byte increments.

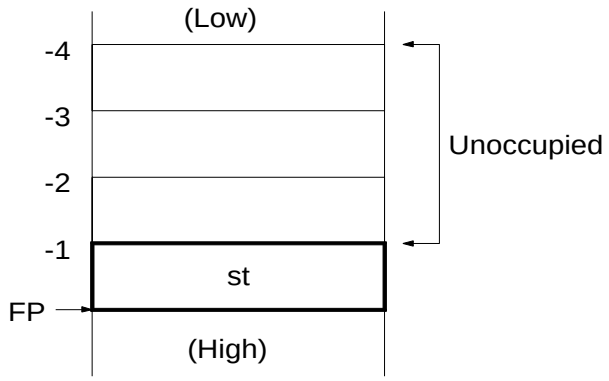
**[Example 3]**

```
struct A st;
sub(st);
```



**[Example 4]**

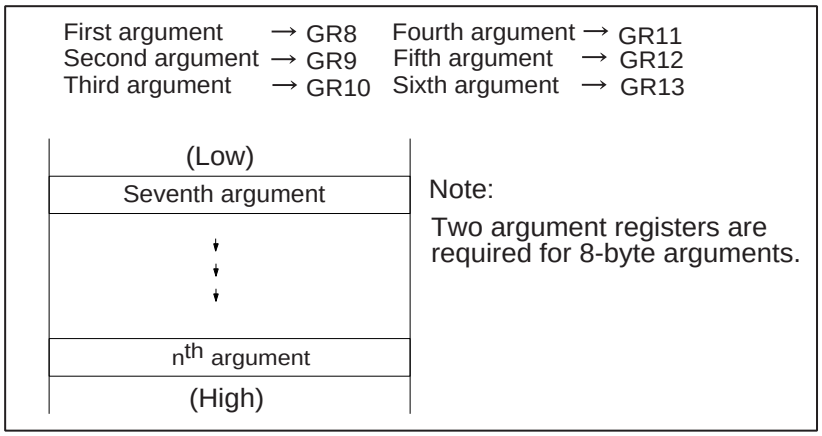
```
struct A{ char a; }st;
```




When a function receiving the variable number of arguments is to be called, the arguments are placed in registers in the same manner as for transfer. The called function stores all of the register-delivered arguments in the argument register save area in the stack. The actual argument area is allocated/deallocated by the caller function, whereas the argument register save area is allocated/deallocated by the called function.

Figure 5.6-2 "fcc935s Command Argument Format Stated in Standard Linkage Regulations" shows the argument format prescribed in the standard linkage regulations.

**Figure 5.6-2 fcc935s Command Argument Format Stated in Standard Linkage Regulations**



## 5.6.3 fcc935s Command Argument Extension Format

**When an argument is to be stored in the stack, its type is converted to an extended type in accordance with the individual argument type. The argument is released by the caller function after control is returned from the called function.**

### ■ fcc935s Command Argument Extension Format

Table 5.6-1 "fcc911 Command Argument Extension Format" lists the argument extension format.

**Table 5.6-1 fcc935s Command Argument Extension Format**

| Actual argument type | Extended type *1 | Stack storage size (byte) |
|---|---|---|
| char | int | 4 |
| signed char | int | 4 |
| unsigned char | int | 4 |
| short | int | 4 |
| unsigned short | int | 4 |
| int | No extension | 4 |
| unsigned int | No extension | 4 |
| long | No extension | 4 |
| unsigned long | No extension | 4 |
| long long | No extension | 8 |
| unsigned long long | No extension | 8 |
| float | Double | 8 |
| double | No extension | 8 |
| long double | No extension | 8 |
| pointer/address | No extension | 4 |
| Structure/union | - | 4 *2 |

*1: The extended type represents an extended type that is provided when no argument type is given. When a prototype declaration is made, it is complied with.

*2: When a structure/union is to be delivered as an argument, the caller copies it to the local variable area and delivers the address of that area.

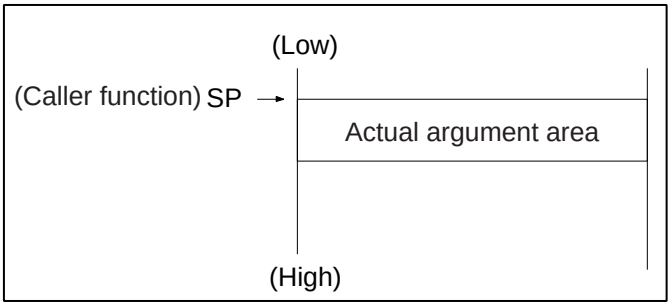
## 5.6.4 fcc935s Command Calling Procedure

**The caller function initiates branching to the called function after argument storage.**

### ■ fcc935s Command Calling Procedure

Figure 5.6-3 "Stack Frame Prevailing at Calling in Compliance with fcc935s Command Standard Linkage Regulations" shows the stack frame prevailing at calling in compliance with the standard linkage regulations.

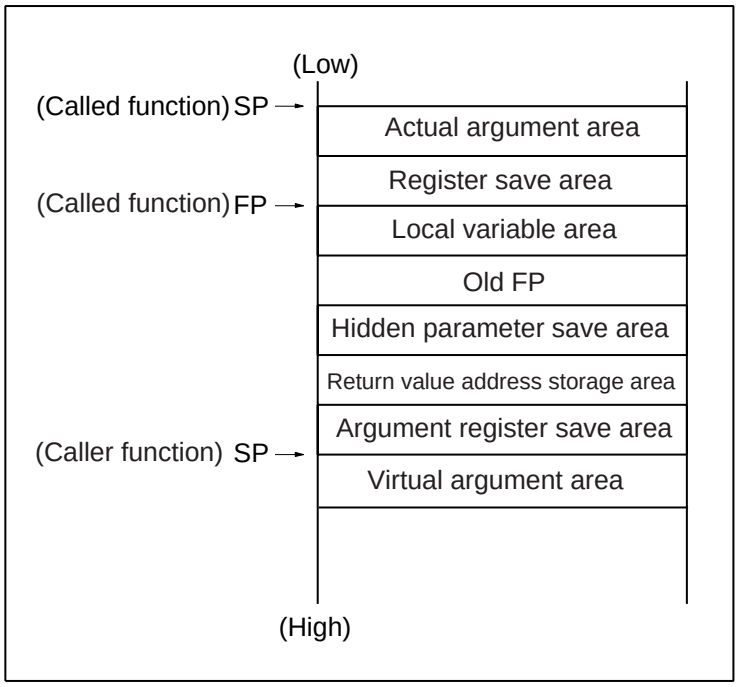
**Figure 5.6-3 Stack Frame Prevailing at Calling in Compliance with fcc935s Command Standard Linkage Regulations**




The called function saves the caller function frame pointer (FP) in the stack and then stores the prevailing stack pointer value in the stack as the new frame pointer value. Subsequently, the local variable area and the caller function register save area are acquired from the stack to save the caller register.

Figure 5.6-4 "Stack Frame Created by Called Function in Compliance with fcc935s Command Standard Linkage Regulations" shows the stack frame that is created by the called function in compliance with the standard linkage regulations.

**Figure 5.6-4 Stack Frame Created by Called Function in Compliance with fcc935s Command Standard Linkage Regulations**



# 5.6.5 fcc935s Command Register

**This section describes the register assurance and register setup regulations in the standard linkage regulations.**

## ■ fcc935s Command Register

The called function assures the following registers of the caller function.

**General-purpose registers GR1, GR2, GR16 to GR31, GR48 to GR63, FR16 to FR31, and FR48 to FR63**

The register assurance is provided when the called function acquires a new area from the stack and saves the register value in that area. However, the contents of registers that has not changed through the function are not saved.If such registers are altered using the ASM statement, subsequent operation is not assured.

## ■ fcc935s Command Register Setup

Table 5.6-2 "Register Regulations for fcc935s Command Function Call and Return Periods" lists the register regulations for function call and return periods.

**Table 5.6-2 Register Regulations for fcc935s Command Function Call and Return Periods**

| Register | Call period | Return period |
|---|---|---|
| GR0 | Zero register | 0 |
| GR1 | Stack pointer (SP) | Call period value guaranteed |
| GR2 | Frame pointer (FP) | Call period value guaranteed |
| GR3 | Argument/return value area address*1 | Return value*2 |
| GR4-GR7 | Not stipulated | Not stipulated |
| GR8, GR9 | Argument | Return value*2 |
| GR10-GR13 | Argument register*1 | Not stipulated |
| GR14, GR15 | Not stipulated | Not stipulated |
| GR16-GR31 | Not stipulated | Call period value guaranteed |
| GR32-GR47 | Not stipulated | Not stipulated |
| GR48-GR63 | Not stipulated | Call period value guaranteed |
| FR0-FR15 | Not stipulated | Not stipulated |
| FR16-FR31 | Not stipulated | Call period value guaranteed |

**Table 5.6-2 Register Regulations for fcc935s Command Function Call and Return Periods (Continued)**

| Register | Call period | Return period |
|---|---|---|
| FR32-FR47 | Not stipulated | Not stipulated |
| FR48-FR63 | Not stipulated | Call period value guaranteed |

*1 There are no rules for unused registers in situations where the argument is smaller than 4 words.

*2 There are no rules for situations where a function without the return value is called or a function with a structure/union type return value is called.

# 5.6.6 fcc935s Command Return Value

**Table 5.6-3 "fcc935s command return value interface stated in standard linkage regulations" lists the return value interface stated in the standard linkage regulations.**

■ **fcc935s Command Return Value**

Table 5.6-3 "fcc935s command return value interface stated in standard linkage regulations" fcc935s command return value interface stated in standard linkage regulations

**Table 5.6-3 fcc935s command return value interface stated in standard linkage regulations**

| Return value type | Return value interface |
|---|---|
| void | None |
| char | GR8 |
| signed char | GR8 |
| unsigned char | GR8 |
| short | GR8 |
| unsigned short | GR8 |
| int | GR8 |
| unsigned int | GR8 |
| long | GR8 |
| unsigned long | GR8 |
| long long | GR8, GR9 |
| unsigned long long | GR8, GR9 |
| float | GR8 |
| double | GR8, GR9 *1 |
| long double | GR8, GR9 *1 |
| Pointer/address | GR8 |
| Structure/union | GR3 *2 |

*1 The 4 high-order bytes of a total of 8 bytes are stored in GR8 and the 4 low-order bytes are stored in GR9.

*2 When a structure/union is used as the return value, the caller function stores the start address of the return value storage area in GR3 and then passes it to the called function. The called function interprets GR3 as the start address of the return value storage area. When this address needs to be saved in memory, the called function allocates the hidden parameter save area and saves the address in that area.

# CHAPTER 6 EXTENDED LANGUAGE SPECIFICATIONS

**This chapter explains about the extended language specifications supported by the compiler. The limitations on compiler translation are also described in this chapter.**


6.1 "Assembler Description Functions"

6.2 "Interrupt Control Functions"

6.3 "I/O Area Access Function"

6.4 "In-line Expansion Specifying Function"

6.5 "Section Name Change Function"

6.6 "Interrupt Level Setup Function"

6.7 "Branch Probability Specification Function"

6.8 "Built-in Function"

6.9 "Predefined Macros"

6.10 "Limitations on Compiler Translation"

6.11 "Re-include Prevention Function"

6.12 "Function for Controlling Instantiation of C++ Template"

# 6.1 Assembler Description Functions

**There are the following two assembler description functions.**

- **asm statement**
- **Pragma instruction**

## ■ Description by asm Statement

When the asm statement is written, the character string literal is expanded as the assembler instruction. This function makes it possible to write the asm statement inside and outside the function.

**[General format]**

_asm (Character string literal);

**[Explanation]**

When the asm statement is written inside the function, the assembler is expanded at the written position.

When the statement is written outside the function, it is expanded as an independent section. Therefore, if the statement is to be written outside the function, be sure to write the section definition pseudo instruction to define the section. If the section is not defined, no subsequent operations will be guaranteed.

When using a general-purpose register within the asm statement in the function during fcc911s command execution, the user is responsible for register saving and restoration. However, the user need not to be conscious of general-purpose registers R0 to R3, R12, and R13 because saving and restoring are performed by the compiler.

When using a general-purpose register within the asm statement in the function during fcc935s command execution, the user is responsible for register saving and restoration. However, the user need not recognize general-purpose registers GR4 to GR7, GR32 to GR47, FR0 to FR15, and FR32 to FR47 because the compiler saves and restores them.

If the asm statement exists in a C/C++ source program, various optimization features are inhibited even when the -O optimization option is specified.

**[fcc911s Command Output Example]**

Input:
```
/* When written inside the function */
extern  int  temp;
sample( ){
        _ _asm("      LDI         #1, R0");
        _ _asm("      LDI:32      #_temp, R12");
        _ _asm("      ST          R0, @R12");
}
/* When written outside the function */
_ _asm("            .SECTION      DATA, DATA, ALIGN=4");
_ _asm("            .GLOBAL  _a");
_ _asm("_a:");
_ _asm("            .RES.B   4");
```

Output:
```
            .SECTION          CODE, CODE, ALIGN=2 );
;-------begin_of_function
            .GLOBAL  _sample
_sample:
            ST          RP, @-SP
            ENTER       #4
            LDI         #1, R0
            LDI:32      #_temp, R12
            ST          R0, @R12
L_sample:
            LEAVE
            LD          @SP+, RP
            RET
            .SECTION          DATA,   DATA,   ALIGN=4
            .GLOBAL  _a

_a:
            .RES.B   4
```

**[fcc935s Command Output Example]**

Input:

```
/* When written inside the function */
extern  int  temp;
sample( ){
        _ _asm("      setlos      #1,gr4");
        _ _asm("      sethi       #hi(_temp),gr5");
        _ _asm("      setlo       #lo(_temp),gr5");
        _ _asm("      st          gr4,@(gr5,gr0)");
}
/* When written outside the function */
_ _asm("            .SECTION      DATA, DATA, ALIGN=8");
_ _asm("            .GLOBAL  _a");
_ _asm("_a:");
_ _asm("            .RES.B   4");
```

Output:

```
        .section            CODE,CODE,align=4
        .global _sample
_sample:
L_20:
        subi        sp,#16,sp
        st          fp,@(sp,gr0)
        mov         sp,fp
        movsg       LR,gr4
        sti         gr4,@(fp,8)
L_21:
        setlos      #1,gr4
        sethi       #hi(_temp),gr5
        setlo       #lo(_temp),gr5
        st          gr4,@(gr5,gr0)

L_22:
        ldi         @(fp,8),gr4
        movgs       gr4,LR
        addi        fp,#16,sp
        ld          @(fp,gr0),fp
        ret
L_23:
        .SECTION      DATA, DATA, ALIGN=8
        .GLOBAL _a
_a:
        .RES.B 4
```

## ■ Description by Pragma Instruction

Expands the description between #pragma asm and #pragma endasm directly as the assembler instruction. This function makes it possible to write the statement inside and outside the function.

**[General format]**

```
#pragma  asm
        Assembler description
#pragma  endasm
```

**[Explanation]**

When the statement is written inside the function, the assembler is expanded at the written position.

When the statement is written outside the function, it is expanded as an independent section. Therefore, if the statement is to be written outside the function, be sure to write the section definition pseudo instruction to define the section. If the section is not defined, no subsequent operations will be guaranteed.

When using a general-purpose register within the asm statement in the function during fcc911 command execution, the user is responsible for register saving and restoration. However, the user need not to be conscious of general-purpose registers R0 to R3, R12, and R13 because saving and restoring are performed by the compiler.

When using a general-purpose register within the asm statement in the function during fcc935s command execution, the user is responsible for register saving and restoration. However, the user need not recognize general-purpose registers GR4 to GR7, GR32 to GR47, FR0 to FR15, and FR32 to FR47 because the compiler saves and restores them.

If during the C source program provided by #pragma asm/endasm exists in the function, various optimization features are inhibited even when the -O optimization option is specified.

**[fcc911s Command Output Example]**

```
Input:    /* When written inside the function */
          extern  int  temp;
          sample( ){
          #pragma     asm
                      LDI         #1, R0
                      LDI:32      #_temp, R12
                      ST          R0, @R12
          #pragma     endasm
          }
          /* When written outside the function */
          #pragma     asm
                      .SECTION          DATA,   DATA,   ALIGN=4
                      .GLOBAL _a
          _a:
                      .RES.B  4
          #pragma     endasm

Output:               .SECTION          CODE,  CODE,  ALIGN=2
          ;-------begin_of_function
                .GLOBAL _sample
          _sample:
                      ST          RP, @-SP
                      ENTER       #4
                      LDI         #1, R0
                      LDI:32      #_temp, R12
                      ST          R0, @R12
          L_sample:
                      LEAVE
                      LD          @SP+, RP
                      RET
                      .SECTION          DATA,   DATA,   ALIGN=4
                      .GLOBAL _a
          _a:
                      .RES.B   4
```

**[fcc935s Command Output Example]**

Input:
```
/* When written inside the function */
extern  int  temp;
sample( ){
#pragma  asm
            setlos          #1,gr4
            sethi           #hi(_temp),gr5
            setlo           #lo(_temp),gr5
            st                      gr4,@(gr5,gr0)
#pragma   endasm
}
/* When written outside the function */
#pragma   asm
            .SECTION        DATA DATA ALIGN=8
            .GLOBAL _a
_a:
            .RES.B      4
#pragma   endasm
```

Output:
```
            .section            CODE,CODE,align=4
            .global _sample
_sample:
L_20:
            subi        sp,#16,sp
            st          fp,@(sp,gr0)
            mov         sp,fp
            movsg       LR,gr4
            sti         gr4,@(fp,8)
L_21:
            setlos      #1,gr4
            sethi       #hi(_temp),gr5
            setlo       #lo(_temp),gr5
            st          gr4,@(gr5,gr0)

L_22:
            ldi         @(fp,8),gr4
            movgs       gr4,LR
            addi        fp,#16,sp
            ld          @(fp,gr0),fp
            ret
L_23:
            .SECTION        DATA, DATA, ALIGN=8
            .GLOBAL _a
_a:
            .RES.B      4
```

# 6.2 Interrupt Control Functions

**Interrupt Control Functions**

- **Interrupt mask setup function**
- **Interrupt mask disable function**
- **Interrupt level setup function**
- **Interrupt function description function**
- **Interrupt vector table generation function**

■ **Interrupt Mask Setup Function**

**[General format]**

void __DI(void);

**[Explanation]**

This function can only be used in the fcc911s command.

Expands the interrupt masking code.

**[Output Example]**

```
Input:  _ _DI( );
fcc911s Command Output:
        ANDCCR  #0xef
```

■ **Interrupt Mask Disable Function**

**[General format]**

void __EI(void);

**[Explanation]**

This function can only be used in the fcc911s command.

Expands the interrupt masking disable code.

**[Output Example]**

```
Input:  _ _EI( );
fcc911s Command Output:
        ORCCR   #0x10
```

## ■ Interrupt Level Setup Function

**[General format]**

void __set_il(int level);

**[Explanation]**

This function can only be used in the fcc911s command.

Expands the code for changing the interrupt level to the specified level.

**[Output Example]**

```
Input:  _ _set_il(2);
fcc911s Command Output:
        STILM   #2
```

## ■ Interrupt Function Description Function

**[General format1]**

__interrupt void Interrupt function (void){ ... }

**[General format2]**

extern __interrupt void Interrupt function (void);

**[Explanation]**

This function can only be used in the fcc911s command.

The interrupt function can be written by specifying the __interrupt type qualifier. Since the interrupt function is called by an interrupt, it is impossible to set up an argument or obtain a return value. If a function declared or defined by the __interrupt type qualifier is called by performing the normal function calling procedure, no subsequent operations will be guaranteed.

**[Output Example]**

```
Input:  _ _interrupt void sample(void){ ... }
fcc911s Command Output:
        _func:
                STM     (R12, R13)
                ST      MDH, @-SP
                ST      MDL, @-SP
                ST      RP, @-SP
                ENTER   #4
                ....
        L_func:
                LEAVE
                LD      @SP+, RP
                LD      @SP+, MDL
                LD      @SP+, MDH
                LDM     (R12, R13)
                RETI
```

## ■ Interrupt Vector Table Generation Function

**[fcc911s command general format]**

#pragma intvect Interrupt function name Vector number

#pragma defvect Interrupt function name

**[Explanation]**

This function can only be used in the fcc911s command.

#pragma intvect generates an interrupt vector table for which the interrupt function is set.

#pragma defvect specifies the default interrupt function to be set for interrupt vectors not specified by #pragma intvect.

The interrupt vector table is generated in an independent section named INTVECT.

All interrupt vector tables must be defined using the same translation unit (file). If #pragma intvect or #pragma defvect is specified using two or more translation units, no subsequent operations will be guaranteed.

The definition cannot be formulated two or more times for the same vector number. However, no error occurs if the definitions are identical.

No value other than an integer constant may be specified as the vector number. Specify a vector number between 0 and 255.

Reset vectors must always be assigned to 0xFFFFC. For this reason, to set TBR to other than 0xFFC00, use the asm statement to define reset vectors separately.

# 6.3 I/O Area Access Function

**The I/O area operation variable can be declared by specifying the __io type qualifier.**

## ■ I/O Area Access Function

**[General format]**

__io Variable declaration;

**[Explanation]**

The fcc911s command enables the declaration of variables operating the I/O area defined between addresses 0x00 and 0xff by specifying the __io type qualifier. The fcc911s command makes variables available up to address 0x3ff, depending on their type. Since a highly efficient dedicated instruction is provided for I/O area access, a higher-speed, more-compact object can be generated. This instruction cannot be used for variables operating an I/O area positioned at addresses higher than 0xff. To define a variable that accesses such an area, use the volatile type qualifier.

The fcc935s command outputs the variables with __io type qualifier specified to the IO section.

The initial value cannot be specified for variables for which the __io type qualifier is specified.

When the specified variable is for a structure or union, it is assumed that all members are positioned in the I/O area. The variable cannot be specified for structure or union members. For the variable for which the __io type qualifier is specified, compilation is conducted on the assumption that the volatile type qualifier is specified.

When the -K NOVOLATILE option is specified, the volatile type qualifier is not assumed to be specified for the variable for which the __io type qualifier is specified.

**[Output Example]**

Input:
```
#pragma section IO=IOA, attr=IO, locate=0x10
    _ _io int a;
void func(void) { a=1;}
```

fcc911s command output:
```
                .SECTION    IOA, IO, LOCATE=H'00000010
                .GLOBAL  _a
        _a:
                .RES.B   4
                .SECTION    CODE,   CODE,   ALIGN=2
        ;-------begin_of_function
                .GLOBAL  _func
        _func:
            ST          RP, @-SP
            ENTER       #4
            LDI         #1, R0
            MOV         R0, R13
            DMOV        R13, @_a
        L_func:
            LEAVE
            LD          @SP+, RP
            RET
```

fcc935s command output:
```
            .section        IOA,IO,locate=0x00000010
            .global _a
            .align 4
        _a:
            .res.b 4

            .section        CODE,CODE,align=4
            .global _func
        _func:
        L_20:
            subi        sp,#16,sp
            st          fp,@(sp,gr0)
            mov         sp,fp
            movsg       LR,gr4
            sti         gr4,@(fp,8)
        L_21:
            setlos      #1,gr4
            sethi       #hi(_a),gr5
            setlo       #lo(_a),gr5
            st          gr4,@(gr5,gr0)
        L_22:
            ldi         @(fp,8),gr4
            movgs       gr4,LR
            addi        fp,#16,sp
            ld          @(fp,gr0),fp
            ret
```

# 6.4 In-line Expansion Specifying Function

**This function specifies the user definition function for in-line expansion. In-line expansion can be specified with the -x option.**

## ■ In-line Expansion Specifying Function

**[General format]**

#pragma inline Function name [, Function name...]

**[Explanation]**

Recursively called functions cannot be subjected to in-line expansion. It should also be noted that functions may not be subjected to in-line expansion depending on asm statement use, structure/union type argument presence, setjmp function calling, and other conditions.

When there are two or more descriptions for the same translation unit or in-line expansion is specified by an option, all the specified function names are valid.

The in-line expansion specifying is invalid if the -O option is not specified.

# 6.5 Section Name Change Function

**This function is used to change the section name or section attribute and sets the section arrangement address.**

## ■ Section Name Change Function

**[General format]**

#pragma section DEFSECT[=NEWNAME][,attr=SECTATTR][,locate=ADDR]

**[Explanation]**

Change the section name to be output by the compiler from DEFSECT and NEWNAME and the section type to SECTATTR.

The locate address may also be specified as ADDR.

For the section name to be output by the compiler, see section 4.1 "Section Structure of fcc911s" or section 5.1 "Section Structure of fcc935s"; for the section type that can be output, refer to the Assembler Manual.

When the locate address is given, it cannot be specified for the section at linking.

**<Caution>**

The #pragma section affects the entire source, regardless of the location. If DEFSECT is specified many times, the last one is valid. If DEFSECT is specified by the -s option, it takes priority over the others.

The EXT_CTOR_DTOR section cannot be specified and its output is fixed.

**[Output Example]**

```
Input:      #pragma section CODE=program,attr=CODE,locate=0xff
            void main(void) { }

fcc911s command output:
                    .SECTION        program, CODE, LOCATE=H'000000FF
                ;-------begin_of_function
                    .GLOBAL _main
                _main:
                    ST          RP, @-SP
                    ENTER       #4
                L_main:
                    LEAVE
                    LD          @SP+, RP
                    RET

fcc935s command output:
                    .section        program, CODE, locate=0x000000ff
                    .global _main
                _main:
                L_20:
                    ret
```

# 6.6 Interrupt Level Setup Function

**This function is used to set the function interrupt level.**

## ■ Interrupt Level Setup Function

**[General format]**

#pragma ilm(NUM)

#pragma noilm

**[Explanation]**

This function can only be used in the fcc911s command.

#pragma ilm specifies the interrupt level for the subsequently defined function.

#pragma noilm clears the interrupt level specifying.

At fcc 911s command, an integer constant between 0 and 31 can be specified in the NUM position. A hexa-decimal, octal, or decimal number can be described.

Although the interrupt level is changed at the beginning of the specified function, remember that the new interrupt level does not revert to the previous level at completion of function execution.

Always specify #pragma ilm and #pragma noilm as a set. Nesting is not possible.

**[Output Example]**

```
Input:    #pragma ilm(1)
          void func(void){}
          #pragma noilm

fcc911s command output:
          _func:
                    STILM     #1
                    ST        RP,@-SP
                    ENTER     #4
          L_func:
                    LEAVE
                    LD        @SP+,RP
                    RET
```

# 6.7 Branch Probability Specification Function

**This function is used to specify a branch probability for a conditional branch.**

## ■ Branch Probability Specification Function

**[General format]**

#pragma statement if true-percentage

**[Explanation]**

This function specifies a probability that the condition for a conditional branch is true.

This function can only be used with the fcc935s command. The function is valid only when instruction scheduling is performed.

#pragma statement if specifies the probability that the conditional expression of the if statement that follows is true. Once an appropriate branch probability is set, the hint bit for a branch instruction is set properly. When the -K gs option is specified, tracing for scheduled instructions is selected accordingly.

true-percentage is the percentage of the probability that the conditional expression is true. An integer from 0 to 100 can be specified for the percentage. A value outside this range or an expression cannot be coded in true-percentage.

**[Output Example]**

```
Input:  int foo(int a) {
        #pragma statement if 20
          if(a)     a=10;
          else      a=20;
          return a;
        }

fcc935s command output:
        _foo:
                cmp       gr8, gr0, icc0
                beq       icc0, #3, L_23     ; The hint bit is "taken"
                setlos.p  #10, gr10
                                             bra       L_25
        L_23:
                setlos    #20, gr10
        L_25:
                mov       gr10, gr8
                ret
```

# 6.8 Built-in Function

**The following built-in functions are available.**

- **__wait_nop**
- **Media instruction built-in function**
- **Integer operation built-in function**

## ■ __wait_nop Built-in Function

**[General format]**

```
void __wait_nop(void);
```

**[Explanation]**

To properly time I/O access and interrupt generation, formerly, the NOP instruction was inserted using the asm statement. However, when such a method is used, the asm statement may occasionally inhibit various forms of optimization and greatly degrade the file object efficiency.

When the __wait_nop() built-in function is written, the compiler outputs one NOP instruction to the function call entry position. If the function call entry is performed a count of times until all the issued NOP instructions are covered, timing control is exercised to minimize the effect on optimization.

**[Output Example]**

```
Input:    void sample(void){__wait_nop( );}

fcc911s Command Output:
        _sample:
                ST      RP, @-SP
                ENTER   #4
                NOP
        L_sample:
                LEAVE
                LD      @SP+, RP
                RET

fcc935s Command Output:
        _sample:
                nop
        L_20:
                ret
```

## ■ Media instruction built-in function

This function can be used for the fcc935s command only.

This function is used to use the media instructions.

The built-in functions are available for each media instruction supported by the FR-V family CPU.

To use the media instruction built-in function, always include the header file (media.h) for media instruction built-in function. When the header file is not included, the function cannot be recognized as the built-in function.

### ■ Integer operation built-in function

This function can be used for the fcc935s command only.

This function is used to use the integer operation instructions supported by the FR-V family CPU.

To use the integer operation built-in function, always include the header file (builtin.h) for integer operation built-in function beforehand. When the header file is not included, the function cannot be recognized as the built-in function.

## 6.8.1 Media Instruction Built-in Function

**This function can be used for the fcc935s command only.**
**This function is used to use the media instructions.**
**The built-in functions are available for each media instruction supported by the FR-V family CPU.**
**To use the media instruction built-in function, always include the header file (media.h) for media instruction built-in function beforehand. When the header file is not included, the function cannot be recognized as the built-in function.**

■ **Media and Logical Operation (Word) Instruction (M Instruction)**

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MAND FRi, FRj, FRk | unsigned long<br>__MAND(unsigned long, unsigned long) | unsigned long a, b, c;<br>a =__MAND(b, c); |
| FRk =FRi & FRj | | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR1<br>MAND FR0, FR1, FR2<br>STFI FR2, @(FP, -4) |
| MOR FRi, FRj, FRk | unsigned long<br>__MOR(unsigned long, unsigned long) | unsigned long a, b, c;<br>a =__MOR(b, c); |
| FRk =FRi \|FRj | | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR1<br>MOR FR0, FR1, FR2<br>STFI FR2, @(FP, -4) |
| MXOR FRi, FRj, FRk | unsigned long<br>__MXOR(unsigned long, unsigned long) | unsigned long a, b, c;<br>a =__MXOR(b, c); |
| FRk =FRi ^FRj | | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR1<br>MXOR FR0, FR1, FR2<br>STFI FR2, @(FP, -4) |
| MNOT FRi, FRk | unsigned long<br>__MNOT(unsigned long) | unsigned long a, b;<br>a =__MNOT(b); |
| FRk =~Fri | | LDFI @(FP, -8), FR0<br>MNOT FR0, FR1<br>STFI FR1, @(FP, -4) |

## ■ Media rotate (Word) instruction (M instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MROTLI FRi, #s6, FRk | unsigned long<br>__MROTLI(unsigned long, int) | unsigned long a, b;<br>a =__MROTLI(b, 2); |
| FRk = left rotate (FRi, s6) | *The second argument must be an immediate value.<br>*The second argument is masked by 0x3F. | LDFI @(FP, -8), FR0<br>MROTLI FR0, #2, FR1<br>STFI FR1, @(FP, -4) |
| MROTRI FRi, #s6, FRk | unsigned long<br>__MROTRI(unsigned long, int) | unsigned long a, b;<br>a =__MROTRI(b,3); |
| FRk = right rotate(FRi, s6) | *The second argument must be an immediate value.<br>*The second argument is masked by 0x3F. | LDFI @(FP,-8), FR0<br>MROTRI FR0, #3, FR1<br>STFI FR1,@(FP, -4) |

## ■ Media Word CUT Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MWCUT FRi, FRj, FRk<br>MWCUTI FRi, #s6, FRk | Unsigned long<br>__MWCUT(unsigned long long, unsign | unsigned long a, c;<br>unsigned long long b;<br>a =__MWCUT(b, c);<br>a =__MWCUT(b, 4); |
| FRk =CUT(FRi, FRj)<br>FRk =CUT(FRi, s6) | *If the second argument is an immediate value, it is masked by 0x3F to generate the mwcuti instruction. | LDDFI @(FP,-12), FR0<br>LDFI @(FP,-16), FR2<br>MWCUT FR0, FR2, FR3<br>STFI FR3,@(FP,-4)<br>LDDFI @(FP,-12), FR0<br>MWCUTI FR0, #4, FR3<br>STFI FR3,@(FP, -4) |

## ■ Media Mean Operation (Halfword Dual) Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MAVEH FRi, FRj, FRk | Unsigned long<br>__MAVEH(unsigned long, unsigned long); | unsigned long a, b, c;<br>a =__MAVEH(b, c); |
| HI(FRk)=(HI(FRi)+HI(FRj))>>1<br>LO(FRk)=(LO(FRi)+LO(FRj))>>1 | | LDFI @(FP,-8), FR0<br>LDFI @(FP,-12), FR1<br>MAVEH FR0, FR1, FR2<br>STFI FR2,@(FP, -4) |

## ■ Media Shift (Halfword Dual) Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MSLLHI FRi, #s6, FRk | unsigned long<br>__MSLLHI(unsigned long, int);<br>*The second argument must be an immediate value.<br>*The second argument is masked by 0x3F. | unsigned long a, b;<br>a =__MSLLHI(b, 4); |
| HI(FRk)=HI(FRi)<<s6<br>LO(FRk)=LO(FRi)<<s6 | | LDFI @(FP,-8), FR0<br>MSLLHI FR0, #4, FR2<br>STFI FR2,@(FP, -4) |
| MSRLHI FRi, #s6, FRk | unsigned long<br>__MSRLHI(unsigned long, int);<br>*The second argument must be an immediate value.<br>*The second argument is masked by 0x3F. | unsigned long a, b;<br>a =__MSRLHI(b, 4); |
| HI(FRk)=HI(FRi)>>s6<br>LO(FRk)=LO(FRi)>>s6 | | LDFI @(FP,-8), FR0<br>MSRLHI FR0, #4, FR2<br>STFI FR2,@(FP, -4) |
| MSRAHI FRi, #s6, FRk | signed long<br>__MSRAHI( signed long, int);<br>*The second argument must be an immediate value.<br>*The second argument is masked by 0x3F. | signed long a, b;<br>a =__MSRAHI(b, 4); |
| HI(FRk)=HI(FRi)>>s6<br>LO(FRk)=LO(FRi)>>s6 | | LDFI @(FP,-8), FR0<br>MSRAHI FR0, #4, FR2<br>STFI FR2,@(FP, -4) |

## ■ Media Saturation Operation (Halfword Dual) Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MSATHS FRi, FRj, FRk | signed long<br>__MSATHS( signed long, signed long); | signed long a, b, c;<br>a =__MSATHS(b, c); |
| HI(FRk)=SSAT(HI(FRi), HI(FRj))<br>LO(FRk)=SSAT(LO(FRi), LO(FRj)) | | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR1<br>MSATHU FR0, FR1, FR2<br>STFI FR2,@(FP, -4) |
| MSATHU FRi, FRj, FRk | unsigned long<br>__MSATHU(unsigned long, unsigned long); | unsigned long a, b, c;<br>a =__MSATHS(b, c); |
| HI(FRk)=USAT(HI(FRi), HI(FRj))<br>LO(FRk)=USAT(LO(FRi), LO(FRj)) | | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR1<br>MSATHU FR0, FR1, FR2<br>STFI FR2,@(FP, -4) |

## ■ Media Dual Saturation Addition and Subtraction (Halfword Dual) Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MADDHSS FRi, FRj, FRk | signed long<br>__MADDHSS( signed long, signed long); | signed long a, b, c;<br>a =_MADDHSS(b, c); |
| HI(FRk)=SSAT(HI(FRi)+HI (FRj))<br>LO(FRk)=SSAT(LO(FRi)+LO(FRj)) | | LDFI @(FP,-8), FR0<br>LDFI @(FP,-12), FR1<br>MADDHSS FR0, FR1, FR2<br>STFI FR2,@(FP,-4) |
| MADDHUS FRi, FRj, FRk | unsigned long<br>__MADDHUS(unsigned long, unsigned long); | unsigned long a, b, c;<br>a =_MADDHUS(b, c); |
| HI(FRk)=USAT(HI(FRi)+HI (FRj))<br>LO(FRk)=USAT(LO(FRi)+ LO(FRj)) | | LDFI @(FP,-8), FR0<br>LDFI @(FP,-12), FR1<br>MADDHUS FR0, FR1, FR2<br>STFI FR2,@(FP, -4) |
| MSUBHSS FRi, FRj, FRk | signed long<br>__MSUBHSS( signed long, signed long); | signed long a, b, c;<br>a =_MSUBHSS(b, c); |
| HI(FRk)=SSAT(HI(FRi)-HI(FRj))<br>LO(FRk)=SSAT(LO(FRi)-LO(FRj)) | | LDFI @(FP,-8), FR0<br>LDFI @(FP,-12), FR1<br>MSUBHSS FR0, FR1, FR2<br>STFI FR2,@(FP, -4) |

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MSUBHUS FRi, FRj, FRk | unsigned long<br>__MSUBHUS(unsigned long, unsigned long); | unsigned long a, b, c;<br>a =_MSUBHUS(b, c); |
| HI(FRk)=USAT(HI(FRi)-HI(FRj))<br>LO(FRk)=USAT(LO(FRi)-LO(FRj)) | | LDFI @(FP,-8), FR0<br>LDFI @(FP,-12), FR1<br>MSUBHUS FR0, FR1, FR2<br>STFI FR2,@(FP, -4) |

## ■ Media Dual Multiplication (Halfword Dual) Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MMULHS FRi, FRj, ACCk | void<br>__MMULHS(int, signed long, signed long); | signed long b,c;<br>_MMULHS(ACC0, b, c); |
| ACCGk :ACCk=HI(FRi)*HI(FRj)<br>ACCGk+1:ACCk+1=LO(FRi)*LO(FRj) | *The first argument must be ACC0, ACC2, ACC4, or ACC6.<br>*ACC0, ACC2, ACC4, and ACC6 are list constants defined in media.h. | LDFI @(FP,-8), FR0<br>LDFI @(FP,-12), FR1<br>MMULHS FR0, FR1, ACC0 |
| MMULHU FRi, FRj, ACCk | void<br>__MMULHU(int, unsigned long, unsigned long) | unsigned long b,c;<br>_MMULHU(ACC2, b, c); |
| ACCGk :ACCk =HI(FRi)*HI(FRj)<br>ACCGk+1:ACCk+1=LO(FRi)*LO(FRj) | *The first argument must be ACC0, ACC2, ACC4, or ACC6.<br>*ACC0, ACC2, ACC4, and ACC6 are list constants defined in media.h. | LDFI @(FP,-8), FR0<br>LDFI @(FP,-12), FR1<br>MMULHS FR0, FR1, ACC2 |

## ■ Media Dual Cross Multiplication (Halfword Dual) Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MMULXHS FRi, FRj, ACCk | void<br>__MMULXHS(int, signed long, signed long); | signed long b, c;<br>_MMULXHS(ACC4, b, c); |
| ACCGk:ACCk=HI(FRi)*LO(FRj)<br>ACCGk+1:ACCk+1=LO(FRi)*HI(FRj) | *The first argument must be ACC0, ACC2, ACC4, or ACC6.<br>*ACC0, ACC2, ACC4, and ACC6 are list constants defined in media.h. | LDFI @(FP,-8), FR0<br>LDFI @(FP,-12), FR1<br>MMULXHS FR0, FR1, ACC4 |

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MMULXHU FRi, FRj, ACCk | void<br>__MMULXHU(int, unsigned long, unsigned long); | unsigned long b, c;<br>_MMULXHU(ACC6, b, c); |
| ACCGk :ACCk =HI(FRi)*LO(FRj)<br>ACCGk+1:ACCk+1 =LO(FRi)*HI(FRj) | *The first argument must be ACC0, ACC2, ACC4, or ACC6.<br>*ACC0, ACC2, ACC4, and ACC6 are list constants defined in media.h. | LDFI @(FP,-8), FR0<br>LDFI @(FP,-12), FR1<br>MMULXHU FR0, FR1, ACC6 |

## ■ Media Dual Multiplication and Addition Operation (Halfword Dual) Instruction (M instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MMACHS FRi, FRj, ACCk | void<br>__MMACHS(int, signed long, signed long); | signed long b, c;<br>_MMACHS(ACC0, b, c); |
| ACCGk:ACCk+=HI(FRi)*LO(FRj)<br>ACCGk+1:ACCk+1+=LO(FRi)*HI(FRj) | *The first argument must be ACC0, ACC2, ACC4, or ACC6.<br>*ACC0, ACC2, ACC4, and ACC6 are list constants defined in media.h. | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR1<br>MMACHS FR0, FR1, ACC0 |
| MMACHS FRi, FRj, ACCk | void __MMACHU(int, unsigned long, unsigned long); | unsigned long b, c;<br>_MMACHU(ACC2, b, c); |
| ACCGk :ACCk+ =HI(FRi)*LO(FRj)<br>ACCGk+1:ACCk+1+=LO(FRi)*HI(FRj) | *The first argument must be ACC0, ACC2, ACC4, or ACC6.<br>*ACC0, ACC2, ACC4, and ACC6 are list constants defined in media.h. | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR1<br>MMACHU FR0, FR1, ACC2 |

## ■ Media Dual Multiplication and Subtraction Operation (Halfword Dual) Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MMRDHS FRi, FRj, ACCk | void<br>__MMRDHU(int, unsigned long, unsigned long); | signed long b, c;<br>_MMRDHS(ACC4, b, c); |

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| ACCGk :ACCk- =HI(FRi)*LO(FRj)<br>ACCGk+1 :ACCk+1- =LO(FRi)*HI(FRj) | *The first argument must be ACC0, ACC2, ACC4, or ACC6.<br>*ACC0, ACC2, ACC4, and ACC6 are list constants defined in media.h. | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR1<br>MMRDHS FR0, FR1, ACC4 |
| MMRDHU FRi, FRj, ACCk | void<br>__MMRDHU(int, unsigned long, unsigned long); | unsigned long b, c;<br>_MMRDHU(ACC6, b, c); |
| ACCGk :ACCk-=HI(FRi)*LO(FRj)<br>ACCGk+1:ACCk+1-=LO(FRi)*HI(FRj) | *The first argument must be ACC0, ACC2, ACC4, or ACC6.<br>*ACC0, ACC2, ACC4, and ACC6 are list constants defined in media.h. | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR1<br>MMRDHU FR0, FR1, ACC6 |

## ■ Media Quad Saturation Addition and Subtraction (Halfword Quad) Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MQADDHSS FRi, FRj, FRk | signed long long<br>__MQADDHSS(signed long long, signed long long); | signed long long a, b, c;<br>a =_MQADDHSS(b, c); |
| HI(FRk)=SSAT(HI(FRi)+HI (FRj ))<br>LO(FRk )=SSAT(LO(FRi) +LO(FRj ))<br>HI(FRk+1)=SSAT(HI(FRi+1)+HI(FRj+1))<br>LO(FRk+1)=SSAT(LO(FRi+1)+LO(FRj+1)) | | LDDFI @(FP, -16), FR0<br>LDDFI @(FP, -24), FR2<br>MQADDHSS FR0, FR2, FR4<br>STDFI FR4,@(FP, -8) |
| MQADDHUS FRi, FRj, FRk | unsigned long long<br>__MQADDHUS( unsigned long long,<br>unsigned long long); | unsigned long long a, b, c;<br>a =_MADDHUS(b, c); |
| HI(FRk )=USAT(HI(FRi) +HI(FRj ))<br>LO(FRk )=USAT(LO(FRi) +LO(FRj ))<br>HI(FRk+1)=USAT(HI(FRi+1)+HI(FRj+1))<br>LO(FRk+1)=USAT(LO(FRi+1)+LO(FRj+1)) | | LDDFI @(FP, -16), FR0<br>LDDFI @(FP, -24), FR2<br>MQADDHUS FR0, FR2, FR4<br>STDFI FR4,@(FP, -8) |
| MQSUBHSS FRi, FRj, FRk | signed long long<br>__MQSUBHSS( signed long long,<br>signed long long); | signed long long a, b, c;<br>a =_MSUBHSS(b, c); |
| HI(FRk )=SSAT(HI(FRi )-HI(FRj ))<br>LO(FRk )=SSAT(LO(FRi )-LO(FRj ))<br>HI(FRk+1)=SSAT(HI(FRi+1)-HI(FRj+1))<br>LO(FRk+1)=SSAT(LO(FRi+1)-LO(FRj+1)) | | LDDFI @(FP, -16), FR0<br>LDDFI @(FP, -24), FR2<br>MQSUBHSS FR0, FR2, FR4<br>STDFI FR4,@(FP, -8) |
| MQSUBHUS FRi, FRj, FRk | unsigned long long<br>__MSUBHUS(unsigned long long,<br>unsigned long long); | unsigned long long a, b, c;<br>a =_MSUBHUS(b, c); |
| HI(FRk )=USAT(HI(FRi )-HI(FRj ))<br>LO(FRk )=USAT(LO(FRi )-LO(FRj ))<br>HI(FRk+1)=USAT(HI(FRi+1)-HI(FRj+1))<br>LO(FRk+1)=USAT(LO(FRi+1)-LO(FRj+1)) | | LDDFI @(FP, -16), FR0<br>LDDFI @(FP, -24), FR2<br>MQSUBHUS FR0, FR2, FR4<br>STDFI FR4,@(FP, -8) |

## ■ Media Quad Multiplication (Halfword Quad) Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MQMULHS FRi, FRj, ACCk | void<br>__MQMULHS(int, signed long long,<br>signed long long); | signed long long b, c;<br>_MQMULHS(ACC0, b, c); |
| ACCGk :ACCk =HI(FRi )*HI(FRj )<br>ACCGk+1:ACCk+1 =LO(FRi)*LO(FRj )<br>ACCGk+2:ACCk+2=HI(FRi+1)*HI(FRj+1)<br>ACCGk+3:ACCk+3=LO(FRi+1)*LO(FRj+1) | *The first argument must be ACC0 or ACC4.<br>*ACC0 and ACC4 are list constants defined in media.h. | LDDFI @(FP,-16), FR0<br>LDDFI @(FP,-24), FR2<br>MQMULHS FR0, FR2, ACC0 |
| MQMULHU FRi,FRj,ACCk | void<br>__MQMULHU(int, unsigned long long,<br>unsigned long long); | unsigned long long b, c;<br>_MQMULHU(ACC4, b, c); |
| ACCGk :ACCk =HI(FRi) *HI(FRj )<br>ACCGk+1:ACCk+1 =LO(FRi)*LO(FRj )<br>ACCGk+2:ACCk+2 =HI(FRi+1)*HI(FRj+1)<br>ACCGk+3:ACCk+3=LO(FRi+1)*LO(FRj+1) | *The first argument must be ACC0 or ACC4.<br>*ACC0 and ACC4 are list constants defined in media.h. | LDDFI @(FP, -16), FR0<br>LDDFI @(FP, -24), FR2<br>MQMULHU FR0, FR2, ACC4 |

## ■ Media Quad Cross Multiplication (Halfword Quad) Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MQMULXHS FRi, FRj, ACCk | void<br>__MQMULXHS(int, signed long long,<br>signed long long); | signed long long b, c;<br>_MMULXHS(ACC0, b, c); |
| ACCGk :ACCk =HI(FRi) *LO(FRj )<br>ACCGk+1:ACCk+1=LO(FRi)*HI(FRj)<br>ACCGk+2:ACCk+2=HI(FRi+1)*LO(FRj+1)<br>ACCGk+3:ACCk+3 =LO(FRi+1)*HI(FRj+1) | *The first argument must be ACC0 or ACC4.<br>*ACC0 and ACC4 are list constants defined in media.h. | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR2<br>MMULXHS FR0, FR2, ACC0 |
| MQMULXHU FRi, FRj, ACCk | void<br>__MQMULXHU(int, unsigned long long<br>, unsigned long long); | unsigned long long b, c;<br>_MMULXHU(ACC4, b, c); |
| ACCGk :ACCk =HI(FRi) *LO(FRj )<br>ACCGk+1:ACCk+1 =LO(FRi)*HI(FRj)<br>ACCGk+2:ACCk+2=HI(FRi+1)*LO(FRj+1)<br>ACCGk+3:ACCk+3 =LO(FRi+1)*HI(FRj+1) | *The first argument must be ACC0 or ACC4.<br>*ACC0 and ACC4 are list constants defined in media.h. | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR2<br>MMULXHU FR0, FR2, ACC4 |

## ■ Media Quad Multiplication and Addition Operation (Halfword Quad) Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MQMACHS FRi, FRj, ACCk | void<br>__MQMACHS(int, signed long long,<br>signed long long); | signed long long b, c;<br>_MMACHS(ACC0, b, c); |
| ACCGk :ACCk +=HI(FRi )*LO(FRj )<br>ACCGk+1:ACCk+1 +=LO(FRi )*HI(FRj )<br>ACCGk+2:ACCk+2<br>+=HI(FRi+1)*HI(FRj+1)<br>ACCGk+3:ACCk+3<br>+=LO(FRi+1)*LO(FRj+1) | *The first argument must be ACC0 or ACC4.<br>*ACC0 and ACC4 are list constants defined in media.h. | LDDFI @(FP,-16), FR0<br>LDDFI @(FP,-24), FR2<br>MQMACHS FR0, FR2, ACC0 |
| MQMACHU FRi, FRj, ACCk | void<br>__MQMACHU(int, unsigned long long,<br>unsigned long long); | unsigned long long b, c;<br>_MQMACHU(ACC4, b, c); |
| ACCGk :ACCk +=HI(FRi) *LO(FRj )<br>ACCGk+1:ACCk+1 +=LO(FRi )*HI(FRj )<br>ACCGk+2:ACCk+2+=HI(FRi+1)*HI(FRj+1)<br>ACCGk+3:ACCk+3+=LO(FRi+1)*LO(FRj+1) | *The first argument must be ACC0 or ACC4.<br>*ACC0 and ACC4 are list constants defined in media.h. | LDDFI @(FP, -16), FR0<br>LDDFI @(FP, -24), FR2<br>MQMACHU FR0, FR2, ACC4 |

## ■ Media Dual Complex Multiplication and Addition Operation (Halfword Dual) Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MCPXRS FRi, FRj, ACCk | void<br>__MCPXRS(int, signed long, signed long); | signed long b, c;<br>_MCPXRS(ACC1, b, c); |
| ACCGk:ACCk =HI(Fri)*LO(FRj)<br>-LO(Fri)*LO(FRj) | *The first argument must be ACC0 to ACC7.<br>*ACC0 to ACC7 are list constants defined in media.h. | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR2<br>MCPXRS FR0, FR2, ACC1 |
| MCPXRU FRi, FRj, ACCk | void<br>__MCPXRU(int, unsigned long, unsigned long); | unsigned long b, c;<br>_MCPXRU(ACC3, b, c); |
| ACCGk:ACCk =HI(Fri)*LO(FRj)<br>-LO(Fri)*LO(FRj) | *The first argument must be ACC0 to ACC7.<br>*ACC0 to ACC7 are list constants defined in media.h. | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR2<br>MCPXRU FR0, FR2, ACC3 |

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MCPXIS FRi, FRj, ACCk | void<br>__MCPXIS(int, signed long, signed long); | signed long b, c;<br>_MCPXIS(ACC5, b, c); |
| ACCGk:ACCk =HI(Fri)*LO(FRj)<br>+LO(Fri)*LO(FRj) | *The first argument must be ACC0 to ACC7.<br>*ACC0 to ACC7 are list constants defined in media.h. | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR2<br>MCPXIS FR0, FR2, ACC5 |
| MCPXIU FRi, FRj, ACCk | void<br>__MCPXIU(int, unsigned long, unsigned long); | unsigned long b, c;<br>_MCPXIU(ACC7, b, c); |
| ACCGk:ACCk =HI(Fri)*LO(FRj)<br>+LO(Fri)*LO(FRj) | *The first argument must be ACC0 to ACC7.<br>*ACC0 to ACC7 are list constants defined in media.h. | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR2<br>MCPXIU FR0, FR2, ACC7 |

## ■ Media Quad Complex Product Sum Operation (Halfword Quad) Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MQCPXRS FRi, FRj, ACCk | void<br>__MQCPXRS(int, signed long long,<br>signed long long); | signed long long b, c;<br>_MQCPXRS(ACC0, b, c); |
| ACCGk :ACCk =HI(FRi )*LO(FRj )<br>-LO(FRi )*LO(FRj )<br>ACCGk+1:ACCk+1<br>=HI(FRi+1)*LO(FRj+1)<br>-LO(FRi+1)*LO(FRj+1) | *The first argument must be ACC0, ACC2, ACC4, or ACC6.<br>*ACC0, ACC2, ACC4, and ACC6 are list constants defined in media.h. | LDDFI @(FP, -16), FR0<br>LDDFI @(FP, -24), FR2<br>MQCPXRS FR0, FR2, ACC0 |
| MQCPXRU FRi, FRj, ACCk | void<br>__MQCPXRU(int, unsigned long long,<br>unsigned long long); | unsigned long long b, c;<br>_MQCPXRU(ACC2, b, c); |
| ACCGk :ACCk =HI(FRi )*LO(FRj )<br>-LO(FRi )*LO(FRj )<br>ACCGk+1:ACCk+1<br>=HI(FRi+1)*LO(FRj+1)<br>-LO(FRi+1)*LO(FRj+1) | *The first argument must be ACC0, ACC2, ACC4, or ACC6.<br>*ACC0, ACC2, ACC4, and ACC6 are list constants defined in media.h. | LDDFI @(FP, -16), FR0<br>LDDFI @(FP, -24), FR2<br>MQCPXRU FR0, FR2, ACC2 |
| MQCPXIS FRi, FRj, ACCk | void<br>__MQCPXIS(int, signed long long,<br>signed long long); | signed long long b,c;<br>_MQCPXIS(ACC4, b, c); |

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| ACCGk :ACCk =HI(FRi )*LO(FRj )<br>+LO(FRi )*LO(FRj )<br>ACCGk+1:ACCk+1<br>=HI(FRi+1)*LO(FRj+1)<br>+LO(FRi+1)*LO(FRj+1) | *The first argument must be ACC0, ACC2, ACC4, or ACC6.<br>*ACC0, ACC2, ACC4, and ACC6 are list constants defined in media.h. | LDDFI @(FP, -16), FR0<br>LDDFI @(FP, -24), FR2<br>MQCPXIS FR0, FR2, ACC4 |
| MQCPXIU FRi, FRj, ACCk | void<br>__MQCPXIU(int, unsigned long long,<br>unsigned long long); | unsigned long long b, c;<br>_MQCPXIU(ACC6, b, c); |
| ACCGk :ACCk =HI(FRi )*LO(FRj )<br>+LO(FRi )*LO(FRj )<br>ACCGk+1:ACCk+1<br>=HI(FRi+1)*LO(FRj+1)<br>+LO(FRi+1)*LO(FRj+1) | *The first argument must be ACC0, ACC2, ACC4, or ACC6.<br>*ACC0, ACC2, ACC4, and ACC6 are list constants defined in media.h. | LDDFI @(FP, -16), FR0<br>LDDFI @(FP, -24), FR2<br>MQCPXIU FR0, FR2, ACC6 |

## ■ Media CUT Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MCUT ACCi,FRj,FRk<br>MCUTI ACCi,#s6,FRk | unsigned long<br>__MCUT(int, unsigned long); | unsigned long a, c;<br>a =_MCUT(ACC0, c);<br>a =_MCUT(ACC1, 4); |
| FRk =CUT(ACCGi:ACCi,FRj)<br>FRk =CUT(ACCGi:ACCi,s6) | *The first argument must be ACC0 to ACC7.<br>*ACC0 to ACC7 are list constants defined in media.h.<br>*If the second argument is an immediate value, it is masked by 0x3F. | LDFI @(FP,-8), FR0<br>MCUT ACC0,FR0, FR1<br>STFI FR1,@(FP, -4)<br>MCUTI ACC1,#4, FR1<br>STFI FR1,@(FP, -4) |
| MCUTSS ACCi,FRj,FRk<br>MCUTSSI ACCi,#s6,FRk | unsigned long<br>__MCUTSS(int, signed long); | unsigned long a;<br>signed long c;<br>a =_MCUTSS(ACC0, c);<br>a =_MCUTSS(ACC1, 4); |
| FRk=SSAT(CUT(ACCGi:ACCi,FRj))<br>FRk =SSAT(CUT(ACCGi:ACCi,s6)) | *The first argument must be ACC0 to ACC7.<br>*ACC0 to ACC7 are list constants defined in media.h.<br>*If the second argument is an immediate value, it is masked by 0x3F. | LDFI @(FP, -8), FR0<br>MCUTSS ACC0, FR0, FR1<br>STFI FR1,@(FP, -4)<br>MCUTSSI ACC1, #4, FR1<br>STFI FR1,@(FP, -4) |

## ■ Media Halfword Expand Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MEXPDHW FRi, #s6, FRk | unsigned long<br>__MEXPDHW(unsigned long, int);<br>*The second argument must be an immediate value.<br>*The second argument is masked by 0x3F. | unsigned long a, b;<br>a =_MEXPDHW(b, 4); |
| HI(FRk)=EXPAND(FRi, s6)<br>LO(FRk)=EXPAND(FRi, s6) | | LDFI @(FP, -8), FR0<br>MEXPDHW FR0, #4, FR1<br>STFI FR1,@(FP, -4) |
| MEXPDHD FRi, #s6, FRk | unsigned long long<br>__MEXPDHD(unsigned long, int);<br>*The second argument must be an immediate value.<br>*The second argument is masked by 0x3F. | unsigned long b;<br>unsigned long long a;<br>a =_MEXPDHD(b, 4); |
| HI(FRk )=EXPAND(FRi, s6)<br>LO(FRk )=EXPAND(FRi, s6)<br>HI(FRk+1)=EXPAND(FRi, s6)<br>LO(FRk+1)=EXPAND(FRi, s6) | | LDFI @(FP, -8), FR0<br>MEXPDHW FR0, #4, FR2<br>STDFI FR2,@(FP, -4) |

## ■ Media Halfword Pack/Unpack Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MPACKH FRi, FRj, FRk | unsigned long<br>__MPACKH(unsigned short, unsigned short); | unsigned long a;<br>unsigned short b, c;<br>a =_MPACKH(b, c); |
| HI(FRk)=LO(FRi)<br>LO(FRk)=LO(FRj) | | LDHFI @(FP, -6), FR0<br>LDHFI @(FP, -8), FR1<br>MPACKH FR0, FR1, FR2<br>STFI FR2,@(FP, -4) |
| MUNPACKH Fri, FRk | unsigned long long<br>__MUNPACKH(unsigned long); | Unsigned long long a;<br>Unsigned long b;<br>a =_MUNPACKH(b); |
| HI(FRk )=HI(FRi)<br>LO(FRk )=HI(FRi)<br>HI(FRk+1)=LO(FRi)<br>LO(FRk+1)=LO(FRi) | | LDFI @(FP, -12), FR0<br>MUNPACKH FR0, FR1<br>STDFI FR1,@(FP, -8) |

## ■ Media Halfword Pack (Dual) Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MDPACKH FRi, FRj, FRk | unsigned long long<br>__MDPACKH(unsigned short, unsigned short,<br>unsigned short, unsigned short); | unsigned long long a;<br>unsigned short b, c, d, e;<br>a =_MDPACKH(b, c, d, e); |
| HI(FRk )=LO(FRi )<br>LO(FRk )=LO(FRj )<br>HI(FRk+1)=LO(FRi+1)<br>LO(FRk+1)=LO(FRj+1) | | LDHFI @(FP, -10), FR0<br>LDHFI @(FP, -12), FR1<br>LDHFI @(FP, -14), FR2<br>LDHFI @(FP, -16), FR3<br>MDPACKH FR0, FR2, FR4<br>STDFI FR4,@(FP, -8) |

## ■ Media Byte-Halfword Conversion Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MBTOH FRi,FRk | unsigned long long<br>__MBTOH(unsigned long); | unsigned long long a;<br>unsigned long b;<br>a =_MBTOH(b); |
| HI(FRk)=(unsignedshort)BYTE3(FRi)<br>LO(FRk )=(unsigned short)BYTE2(FRi)<br>HI(FRk+1)=(unsigned short)BYTE1(FRi)<br>LO(FRk+1)=(unsigned short)BYTE0(FRi) | | LDFI @(FP, -12), FR0<br>MBTOH FR0, FR2<br>STDFI FR2,@(FP, -8) |
| MHTOB FRi,FRk | unsigned long<br>__MHTOB(unsigned long long); | unsigned long a;<br>unsigned long long b;<br>a =_MHTOB(b); |
| BYTE1(FRk)=(unsigned char)USAT(HI(FRi ))<br>BYTE0(FRk)=(unsigned char)USAT(LO(FRi ))<br>BYTE3(FRk)=(unsigned char)USAT(HI(FRi+1))<br>BYTE2(FRk)=(unsigned char)USAT(LO(FRi+1)) | | LDDFI @(FP, -12), FR0<br>MHTOB FR0, FR2<br>STFI FR2,@(FP, -4) |

## ■ Media Accumulator Clear Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MCLRACC ACCk,#A | void<br>__MCLRACC(int);<br>*The first argument must be ACC0 to ACC7.<br>*ACC0 to ACC7 are list constants defined in media.h.<br><br>void<br>_MCLRACCA(void); | _MCLRACC(ACC0);<br>_MCLRACCA(); |
| IF(#A==0){<br>ACCGk:ACCk = 0<br>}ELSE{<br>FOR(k=0; k<ACCmax; k++){<br>ACCGk:ACCk = 0;<br>}<br>} | | MCLRACC ACC0, #0<br>MCLRACC ACC0, #1 |

## ■ Media Accumulator Read/Write Instruction (M Instruction)

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MRDACC ACCi, FRk | unsigned long<br>__MRDACC(int);<br>*The first argument must be ACC0 to ACC7.<br>*ACC0 to ACC7 are list constants defined in media.h. | unsigned long a;<br>a =_MRDACC(ACC1); |
| FRk =ACCi | | MRDACC ACC1, FR1<br>STFI FR1,@(FP, -4) |
| MWTACC FRi, ACCk | void<br>__MWTACC(int, unsigned long);<br>*The first argument must be ACC0 to ACC7.<br>*ACC0 to ACC7 are list constants defined in media.h. | unsigned long b;<br>_MRDACC(ACC2,b); |
| ACCk =Fri | | LDFI @(FP,-8),FR0<br>MWTACC FR0,ACC2 |
| MRDACCG ACCGi, FRk | unsigned long<br>__MRDACCG(int);<br>*The first argument must be ACCG to ACCG7.<br>*ACCG0 to ACCG7 are list constants defined in media.h. | unsigned long a;<br>a =_MRDACCG(ACCG1); |
| FRk =ACCGi | | MRDACCG ACCG1,FR1<br>STFI FR1,@(FP,-4) |

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MWTACCG FRi, ACCGk | void<br>__MWTACCG(int, unsigned long);<br>*The first argument must be ACCG to ACCG7.<br>*ACCG0 to ACCG7 are list constants defined in media.h. | unsigned long b;<br>_MRDACCG(ACCG2,b); |
| ACCGk =Fri | | LDFI @(FP,-8),FR0<br>MWTACCG FR0,ACCG2 |

■ **Media Custom Instruction (M Instruction)**

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| Mcop1 FRi,FRj,FRk | unsigned long<br>__Mcop1(unsigned long, unsigned long); | unsigned long a, b, c;<br>a =_Mcop1(b, c); |
| | | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR1<br>Mcop1 FR0, FR1, FR2<br>STFI FR2,@(FP, -4) |
| Mcop2 FRi,FRj,FRk | unsigned long<br>__Mcop2(unsigned long, unsigned long); | unsigned long a, b, c;<br>a =_Mcop2(b, c); |
| | | LDFI @(FP, -8), FR0<br>LDFI @(FP, -12), FR1<br>Mcop2 FR0, FR1, FR2<br>STFI FR2,@(FP, -4) |

■ **Media Trap Instruction (M Instruction)**

| Assembler notation<br>Operation of the instruction | Media built-in function prototype | Usage example<br>Expansion example |
|---|---|---|
| MTRA | void<br>__MTRAP(void); | __MTRAP(); |
| | | MTRAP |

## 6.8.2 Integer Operation Built-in Function

**This function can be used for the fcc935s command only.**
**This function is used to use the integer operation instructions supported by the FR-V family CPU.**
**To use the integer operation built-in function, always include the header file (builtin.h) for integer operation built-in function. When the header file is not included, the function cannot be recognized as the built-in function.**

■ **Integer Operation Built-in Function**

| Function prototype | Usage example<br>Expansion example |
|---|---|
| signed long long<br>__SMUL( signed long, signed long); | long long a;<br>long b, c;<br>a =__SMUL(b, c); |
| | LDI @(FP, -8), GR4<br>LDI @(FP, -12), GR5<br>SMUL GR4, GR5, GR6<br>STI GR6,@(FP, 0)<br>STI GR7,@(FP, -4) |
| unsigned long long<br>__UMUL(unsigned long, unsigned long); | Unsigned long long a;<br>unsigned long b, c;<br>a =_UMUL(b, c); |
| | LDI @(FP, -8), GR4<br>LDI @(FP, -12), GR5<br>UMUL GR4, GR5, GR6<br>STI GR6,@(FP, 0)<br>STI GR7,@(FP, -4) |

# 6.9 Predefined Macros

**Macro names predefined by the compiler are explained below.**

## ■ Macros Stipulated by ANSI Standard

The ANSI standard stipulates the macros listed in Table 6.9-1 "Macros stipulated by the ANSI standard"

**Table 6.9-1 Macros stipulated by the ANSI standard**

| Macro | Description |
|---|---|
| __LINE__ | Defines line number of current source line. |
| __FILE__ | Defines source file name. |
| __DATE__ | Defines source file translation date. |
| __TIME__ | Defines source file translation time. |
| __STDC__ | Macro indicating that the processing system meets requirements. When the -Ja option is specified, 0 is selected as the definition. When the -Jc option is specified, 1 is selected as the definition. |

In addition to the macros listed in Table 6.9-1 "Macros stipulated by the ANSI standard", C++ has the macros listed in Table 6.9-2 "C++ macros (in addition to the macros stipulated by the ANSI standard)".

**Table 6.9-2 C++ macros (in addition to the macros stipulated by the ANSI standard)**

| Macro | Description |
|---|---|
| __cplusplus | 1 is defined. |
| c_plusplus | 1 is defined. Nothing is defined when the -Jc option is specified. |
| __embedded_cplusplus | 1 is defined only when the default -Je option is specified. |

## ■ Macros Predefined by fcc911s Command

The fcc911s command predefines the following macros.

| Macro | Description |
|---|---|
| __COMPILER_FCC911__ | Selects 1 as definition. |
| __CPU_ MB number__ | The MB number specified by the -cpu option is defined. |
| __CPU_FR__ | Selects 1 as definition. |

## ■ Macros predefined by fcc935s command

The fcc935s command predefines the following macros.

| Macro | Description |
|---|---|
| __COMPILER_FCC935__ | 1 is defined. |
| __CPU_MB number__ | The MB number specified by the -cpu option is defined. |
| __CPU_FR500__ | 1 is defined. |

# 6.10 Limitations on Compiler Translation

**Table 6.10-1 "List of Translation Limitations" shows the translation limitations to be imposed when the compiler is used. The table also indicates the minimum ANSI requirements to be met.**

## ■ Limitations on Compiler Translation

**Table 6.10-1 List of Translation Limitations**

| No. | Function | ANSI Standard | Compiler |
|---|---|---|---|
| 1 | Count of nesting levels for a compound statement, repetition control structure, and selection control structure | 15 | infinity |
| 2 | Count of nesting levels for condition incorporation | 8 | infinity |
| 3 | Count of pointers, arrays, and function declarators (any combinations of these) for qualifying one arithmetic type, structure type, union type, or incomplete type in a declaration | 12 | infinity |
| 4 | Count of nests provided by parentheses for one complete declarator | 31 | infinity |
| 5 | Count of nest expressions provided by parentheses for one complete expression | 32 | infinity |
| 6 | Count of valid leading characters of internal identifier or macro name | 31 | infinity |
| 7 | Count of valid leading characters of external identifier | 6 | infinity |
| 8 | Count of external identifiers of one translation unit | 511 | infinity |
| 9 | Count of identifiers having the block valid range in one block | 127 | infinity |
| 10 | Count of macro names that can be simultaneously defined by one translation unit | 1024 | infinity |
| 11 | Count of virtual arguments in one function definition | 31 | infinity |
| 12 | Count of actual arguments for one function call | 31 | infinity |
| 13 | Count of virtual arguments in one macro definition | 31 | infinity |
| 14 | Count of actual arguments in one macro call | 31 | infinity |
| 15 | Maximum count of characters in one logical source line | 509 | infinity |
| 16 | Count of characters in a (linked) byte character string literal or wide-angle character string literal (terminal character included) | 509 | infinity |

**Table 6.10-1 List of Translation Limitations (Continued)**

| No. | Function | ANSI Standard | Compiler |
|---|---|---|---|
| 17 | Count of bytes of one arithmetic unit | 32767 | infinity |
| 18 | Count of nesting levels for #include file | 8 | infinity |
| 19 | Count of case name cards in one switch statement (excluding nested switch statements) | 257 | infinity |
| 20 | Count of members of one structure or union | 127 | infinity |
| 21 | Count of enumerated type constants in one enumerated type | 127 | infinity |
| 22 | Count of structure or union nesting levels for one structure declaration array | 15 | infinity |

The "infinity" in the above table indicates the dependence on the memory size available for the system.

# 6.11 Re-include Prevention Function

**The file can have #pragma once at its beginning to prevent the header file from being re-included.**

## ■ Re-include Prevention Function

The file can have #pragma once at its beginning to prevent the header file from being re-included.

**[Example]**

```
file1.h:
        #pragma once
        #include "file2.h"
file2.h:
        #pragma once
        #include "file1.h"
file3.c:
        #include "file1.h"   /* file1.h file2. h are included one each. */
```

# 6.12 Function for Controlling Instantiation of C++ Template

**#pragma instantiate forces instantiation.**
**#pragma do_not_instantiate does not provide instantiation.**
**#pramga can_instantiate provides instantiation if possible.**

## ■ Function for Controlling Instantiation of C++ Template

Instantiation of a C++ template can be controlled depending on the following settings:

- #pragma instantiate template name
- #pragma do_not_instantiate template name
- #pramga can_instantiate template name

The types that can be specified as template names are given below:

- Template class name: A<int>
- Template class declaration: class A<int>
- Member function name: A<int> : : f
- Static data member name: A<int> : : i
- Static data member declaration: int A<int> : : i
- Member function declaration: void A<int> : : f (int,char)
- Template function declaration: char *f(int,float)

### ❍ #pragma instantiate

The specified template is forcibly instantiated in the module.

Instantiation requires a complete template definition.

### ❍ #pragma do_not_instantiate

The specified template is not instantiated in the module.

### ❍ #pramga can_instantiate

The specified template is instantiated in the module if possible. The possibility of instantiation is determined automatically by the compiler system. Note that when the --no_auto_instantiation option is specified, instantiation is not performed.

**<Caution>**

When executing modules, #pragma instantiate is specified only for one template in one module. In other modules, #pragma do_not_instantiate must be specified.

The default is #pramga can_instantiate. If no specification is made, the object is instantiated automatically by the compiler system.

# CHAPTER 7 EXECUTION ENVIRONMENT

---

**User programs are executed with or without the existence of an operating system. In an environment in which the operating system exists, it is necessary to prepare the setup process suitable for the environment.**
**This chapter describes the user program execution procedure to be performed in an environment where no operating system exists.**

---


7.1 "Execution Process Overview"

7.2 "Startup Routine Creation"

# 7.1 Execution Process Overview

**In an environment where no operating system exists, it is necessary to prepare the startup routine which initiates user program execution.**

## ■ Execution Process Overview

The main functions to be incorporated into the startup routine are as follows.

❍ **Environment initialization necessary for program operation**

This initialization must be described by the assembler and completed before user program execution.

❍ **User program calling**

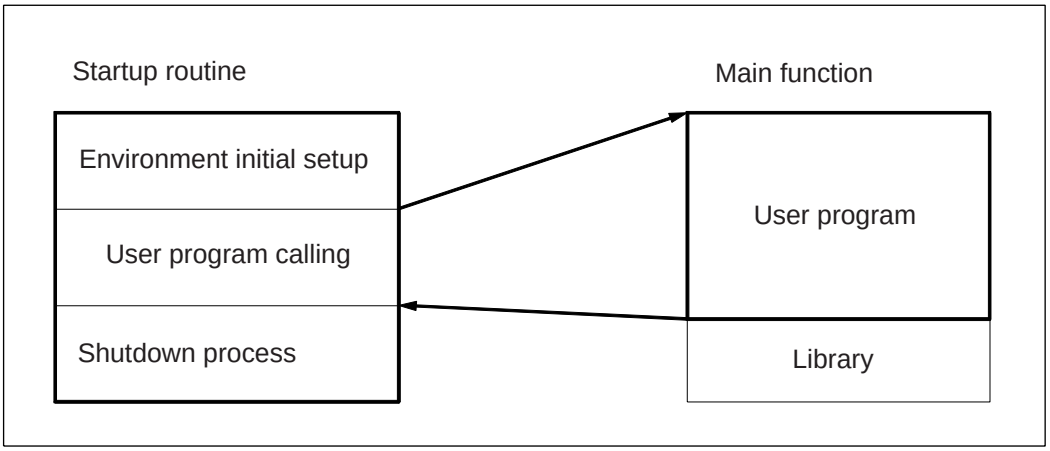
The main function, which is normally used as the function that the startup routine calls in the program start process, is to be called.

❍ **Shutdown process**

After a return from the user program is made, the shutdown process necessary for the system is to be performed to accomplish program termination.

The relationship between the startup routine and user function calling is shown in Figure 7.1-1 "Relationship between Startup Routine and User Function Calling".

**Figure 7.1-1 Relationship between Startup Routine and User Function Calling**




The precautions to be observed in startup routine preparation are described below.

❍ **Stack**

When the user program is executed, the stack is used for return address, argument storage area, automatic variable area, and register saving, etc. The stack must therefore be provided with an adequate space.

❍ **Register**

When the startup routine calls the user program, it is essential that stack pointer setup be completed. The user program operates on the presumption that the stack top is set as the stack

pointer. Further, when the startup routine returns from the user program, the register status is as shown in Table 7.1-1 "fcc911s Command Register Status Prevailing at Return from User Program". This is because the employed interface is the same as for register guarantee at the time of function calling.

For register guarantee, see section 4.6.5 "fcc911 Command Register". If the guarantee of a register is called for by the system while the value of that register is not guaranteed by the user program, it is necessary to guarantee the value by the startup routine to initiate calling.

**Table 7.1-1 fcc911s Command Register Status Prevailing at Return from User Program**

| Register | Value Guarantee at Return |
|---|---|
| R0 to R7 | Not provided |
| R12 to R13 | Not provided |
| R8 and R11 | Provided |
| R14 (FP) | Provided |
| R15 (SP) | Provided |

**Table 7.1-2 fcc935s Command Register Status Prevailing at Return from User Program**

| Register | Value guarantee at return |
|---|---|
| GR1(SP) | Provided |
| GR2(FP) | Provided |
| GR3 - GR15 | Not provided |
| GR16 - GR31 | Provided |
| GR32 - GR47 | Not provided |
| GR48 - GR63 | Provided |
| FR0 - FR15 | Not provided |
| FR16 - FR31 | Provided |
| FR32 - FR47 | Not provided |
| FR48 - FR63 | Provided |

# 7.2 Startup Routine Creation

**The processes necessary for startup routine creation are described below.**

## ■ Startup Routine Creation

1. Register initial setup:

   Set the stack pointer (SP) to the top of the stack (stack top).

2. Data area initialization:

   The C andC++ language specification guarantees the initialization of external variables without the initial value and static variables to 0. Therefore, initialize the DATA section to 0.

3. Initialization data area duplication:

   When incorporating constant data or program into ROM, the data positioned in the ROM area needs to be copied to the RAM area. However, this duplication step is unnecessary if such a data rewrite operation will not performed within the user program. The area to be incorporated into ROM is usually positioned in the INIT section. When incorporation into ROM is specified, the linker automatically generates the following symbols for the specified section name.

   ROM_specified section name

   RAM_specified section name

   The above symbols indicate the ROM and RAM area start addresses, respectively. An example specifying of incorporation into ROM for the INIT section is shown below .

   ```
   > fcc911s -ro ROM=ROM address range  -ra RAM=RAM address range
                                        -sc @INIT=ROM, INIT=RAM .....

   > fcc935s -ro ROM=ROM address range  -ra RAM=RAM address range
                                        -sc @INIT=ROM, INIT=RAM .....
   ```

   For the details of incorporation into ROM, refer to the Linkage Kit Manual

4. Library initial setup:

   When using the libraries, open a file for standard input/output. For details, see section 8.2 "Relationship to Library Incorporating System", Initialization/Termination Process Necessary for Library Use.

5. Initialization unique to C++:

   In the C++ specifications, when external or static objects are used, a constructor must be called followed by the main function. Because four-byte pointers to the main function are stored in the EXT_CTOR_DTOR section, call a constructor sequentially from the lower address of the four addresses in that section.

   In a program requiring normal termination, use the atexit function to register the address of __call_dtors function as the function to be called from the exit function. Then, call the exit function after the end of the main function.

6. User program calling:

   Call the user program.

7. Program shutdown process:

   The close process must be performed for opened files. The normal end and abnormal end processes must be prepared in accordance with the system.

# CHAPTER 8 LIBRARY OVERVIEW

**This chapter outlines the C libraries by describing the organization of files furnished by the libraries and the relationship to the system into which the libraries are incorporated.**

8.1 "File Organization"

8.2 "Relationship to Library Incorporating System"

# 8.1 File Organization

**This section describes the files furnished by the libraries.**

## ■ File Types

The following library files are provided:

### ❍ fcc911s command library files

The four fcc911s command library files below are provided.

- lib911.lib: Standard C library
- lib911if.lib: Simulator/debugger low-level function library
- lib911e.lib: EC++ library
- lib911p.lib: C++ library (having the same contents as lib911e.lib)

### ❍ fcc935s command library files

The four fcc935s command library files below are provided.

- lib935.lib: Standard C library
- lib935if.lib: Simulator/debugger low-level function library
- lib935e.lib: EC++ library
- lib935p.lib: C++ library (having the same contents as lib935e.lib)

## ■ fcc911s Command Library Section Names

Table 8.1-1 "fcc911s Command Library Section Names" show the section names used by the fcc911s command libraries.

**Table 8.1-1 fcc911s Command Library Section Names**

| Section Type | Small |
|---|---|
| Code section | CODE |
| Data section | DATA |
| Initialized section | INIT |
| Constant section | CONST |

## ■ fcc935s Command Library Section Names

Table 8.1-2 "fcc935s Command Library Section Names" show the section names used by the fcc935s command libraries.

**Table 8.1-2 fcc935s Command Library Section Names**

| Section Type | Small |
|---|---|
| Code section | LIBCODE_function name_16 |
| Data section | DATA |

**Table 8.1-2 fcc935s Command Library Section Names (Continued)**

| Section Type | Small |
|---|---|
| Initialized section | INIT |
| Constant section | CONST |

# 8.2 Relationship to Library Incorporating System

**This section describes the relationship between the libraries and library incorporating system.**

■ **System-dependent Processes**

File input/output, memory management, and program termination procedures are the processes dependent on the system. When such system-dependent processes are needed, the libraries issue a call as a low-level function (For the details of low-level functions, see CHAPTER 9 "LIBRARY INCORPORATION").

When using the libraries, prepare such low-level functions in accordance with the system.

■ **Low-level Function (System-dependent Process) Types**

The low-level function types and their roles are summarized below. For the detailed feature descriptions of low-level functions, see section 9.5 "Low-level Function Specifications".

- open: Function for opening a file in the system
- close: Function for closing a file in the system
- read: Function for reading characters from a file
- write: Function for writing characters into a file
- lseek: Function for changing the file position
- isatty: Function for checking whether a file is a terminal file
- sbrk: Function for dynamically acquiring/changing the memory
- _exit: Function for normal program ending
- _abort: Function for abnormal program ending

# CHAPTER 9 LIBRARY INCORPORATION

**This chapter describes the processes and functions to be prepared for library use.**


9.1 "Library Incorporation Overview"

9.2 "Initialization/Termination Process Necessary for Library Use"

9.3 "Low-level Function Types"

9.4 "Standard Library Functions and Required Processes/Low-level Functions"

9.5 "Low-level Function Specifications"

# 9.1 Library Incorporation Overview

**This section outlines library incorporation.**

## ■ Processes and Functions must be prepared for Library Use

File input/output, memory management, and program termination procedures are the processes dependent on the system. Therefore, when such system-dependent processes are needed, such processes are separated from the standard library, and whenever such processes are needed, they will be called as a low-level function. Further, the stream area initialization and other processes are necessary for library use.

The following processes and functions must be prepared for library use.

- Stream area initialization
- Standard input/output and standard error output file open and close processes
- Low-level function creation

At the time of library incorporation, the above processes and functions must be prepared in accordance with the system.

# 9.2 Initialization/Termination Process Necessary for Library Use

**This section describes the initialization/termination process required for library use.**

## ■ Initialization/Termination Process

Some standard library functions require the following processes.

- Steam area initialization
- Standard input/output and standard error output file opening and closing

Detailed in this section (For required functions, see section 9.4 "Standard Library Functions and Required Processes/Low-level Functions)".

## ■ Stream Area Initialization

The _stream_init function initializes the stream area. This function must be called by the startup routine to initialize the stream area.

void _stream_init(void);

## ■ Standard Input/Output and Standard Error Output File Opening and Closing

Because the standard input/output and standard error output do not open or close files during the execution of programs, files must be opened before the main function is called and must be closed when the main function is completed.

Use the startup routine to perform the opening process before main function calling and the closing process after main function execution.

However, the _stream_init function correlates the file numbers 0, 1, and 2 to the stdin, stdout, and stderr streams. Therefore, the opening process need not be performed when the system's standard input, standard output, and standard error output are opened as the file numbers 0, 1, and 2.

If the system's standard input/output and standard error output are not opened or the file numbers do not match, perform the following process to open the system's files.

- freopen("Standard input name", "r", stdin);
- freopen("Standard output name", "w", stdout);
- freopen("Standard error output name", "w", stderr);

Error detection concerning the above process should be conducted as needed.

Further, the file names specified by the open function must be written as the standard input/output and standard error output names.

For the closing process, use the fclose function.

## 9.3 Low-level Function Types

**This section outlines the standard library functions and necessary low-level functions. The standard library functions require the following six types of low-level functions:**

- **File opening and closing (open, close)**
- **Input and output relative to file (read, write)**
- **File position change (lseek)**
- **File inspection (isatty)**
- **Memory area dynamic acquisition (sbrk)**
- **Program abnormal end and normal end (_abort, _exit)**

**The above processes are called from the associated standard libraries to manipulate the system's actual files or exercise program execution control.**

### ■ Low-level Function Types

#### ❍ File Opening and Closing

All functions that open the fopen function and other files call the open function to open an actual file on the system. Similarly, all functions that close the fclose function and other files call the close function to close an actual file on the system.

#### ❍ Input and Output Relative to File

The scanf, printf, and other input/output functions perform input/output operations relative to the system's actual files when the read and write functions are called.

#### ❍ File Position Change

The fseek and other file position manipulation functions acquire or change the system's actual file positions when the lseek function is called.

#### ❍ Memory Area Dynamic Acquisition

The malloc and other memory area dynamic acquisition functions acquire or free specific memory areas when the sbrk function is called.

#### ❍ Program Abnormal End and Normal End

The abort function and exit function call the _abort function and _exit function, respectively, as the termination process.

# 9.4 Standard Library Functions and Required Processes/Low-level Functions

**This section describes the standard library functions and associated initialization/ termination processes and low-level functions.**

## ■ Standard Library Functions and Required Processes/Low-level Functions

Table 9.4-1 "Standard Library Functions and Required Processes/Low-level Functions" lists the standard libraries that use low-level functions, related initialization and termination processes, and low-level functions.

**Table 9.4-1 Standard Library Functions and Required Processes/Low-level Functions**

| Standard Library Function | Low-level Function | | Initialization/Termination Process |
|---|---|---|---|
| assert ()<br>abort ()*1 | open ()<br>read ()<br>lseek ()<br>sbrk () | close ()<br>write ()<br>isatty ()<br>_abort () | Stream area initialization process standard input/output and standard error output opening and closing |
| All stdio.h file operation functions | open ()<br>read ()<br>lseek ()<br>sbrk () | close ()<br>write ()<br>isatty () | Stream area initialization process standard input/output and standard error output opening and closing |
| calloc ()<br>malloc ()<br>realloc ()<br>free () | sbrk () | | |
| exit ()*1 | open ()<br>read ()<br>seek ()<br>sbrk () | close ()<br>write ()<br>isatty ()<br>_exit () | Stream area initialization process standard input/output and standard error output opening and closing |

*1: Then the abort function and exit function are called, they perform the closing process for open files. Therefore, the file manipulation related low-level functions (open, close, read, write, lseek, and sbrk) and stream area initialization and like processes are required.
In a program that is not using a file, the _abort function can be directly called instead of the abort function.
In a program for which function registration is not completed using the atexit function, the _exit function can be directly called instead of the exit function while no file is being used.
In the above instances, file manipulation related low-level function use and stream area initialization are not required.

## 9.5 Low-level Function Specifications

**There are various low-level functions. The open, close, read, write, lseek, and isatty functions provide file processing. The sbrk function provides memory area dynamic allocation. The _exit or _abort function is used to terminate a program by calling the exit or abort function. These low-level functions must be created to suit the system.**

### ■ Low-level Function Specifications

Create the low-level functions in compliance with the specifications stated in this section.

## 9.5.1 open Function

**The open function should be generated according to the specifications in this section.**
**#include <fcntl.h>**
**int open(char *fname, int fmode, int p);**

■ **open Function**

**[Explanation]**

In the mode specified by fmode, open the file having the name specified by fname. For fmode specifying, a combination of the following flags (logical OR) is used. The value 0777 is always delivered as p.

- O_RDONLY: Opens a read-only file.
- O_WRONLY: Opens a write-only file.
- O_RDWR: Opens a read/write file.

  The above three flags are to be exclusively specified.
- O_CREAT: Create this flag when the specified file does not exist. If the specified file already exists, ignore this flag.
- O_TRUNC: If any data remains in the file, discard such data to empty the file.
- O_APPEND: Selects the append mode for file opening. The file position prevailing at the time of opening must be set so as to indicate the end of the file. When writing into a file placed in this mode, start writing at the end of the file without regard to the current file position.
- O_BINARY: Specifies a binary file. Therefore, the file opened must be treated as a binary file. Files for which this is not specified must be treated as text files.

When the name for standard input/output and standard error output, which is determined for system environment setup, is specified as the file name for the first argument, allocate the standard input/output and standard error output to the file to be opened.

**[Return value]**

When file opening is successfully done, the file number must be returned. If file opening is not successfully done, on the other hand, the value -1 must be returned.

## 9.5.2 close Function

**The close function should be generated according to the specifications in this section.**
**int close(int fileno);**

■ **close Function**

**[Explanation]**

The closing process must be performed for the file specified by fileno.

**[Return value]**

When file closing is successfully done, the value 0 must be returned. If file closing is not successfully done, the value -1 must be returned.

## 9.5.3 read Function

**The read function should be generated according to the specifications in this section.**
**int read(int fileno, char *buf, unsigned int size);**

■ **read Function**

**[Explanation]**

From the file specified by fileno, size-byte data must be input into the area specified by buf.

If the text file new line character is other than \n in the system environment at this time, perform setup with the new line character converted to \n by the read function.

**[Return value]**

When the input from the file is successfully done, the input character count must be returned. If the input from the file is not successfully done, the value -1 must be returned. If the file ends in the middle of the input sequence, a value smaller than size can be returned as the input character count.

## 9.5.4 write Function

**The write function should be generated according to the specifications in this section.**
**int write (int fileno, char *buf, unsigned int size);**

■ **write Function**

**[Explanation]**

To the file specified by fileno, size-byte data in the area specified by buf must be outputted. If the file is opened in the append mode, the output must always be appended to the end of the file. If the text file new line character is other than \n in the system environment at this time, the output must be generated with the system environment new line character converted to \n by the write function.

**[Return value]**

When the output to the file is successfully done, the output character count must be returned. If it is not successfully done, the value -1 must be returned.

## 9.5.5 lseek Function

**The lseek function should be generated according to the specifications in this section.**
**#include <unistd.h>**
**long int lseek( int fileno, long int offset, int whence);**

■ **lseek Function**

**[Explanation]**

The file specified by fileno must be moved to a position that is offset bytes away from the position specified by whence. The file position is determined according to the byte count from the beginning of the file. The following three positions are to be specified by whence.

- SEEK_CUR: Adds the offset value to the current file position.
- SEEK_END: Adds the offset value to the end of the file.
- SEEK_SET: Adds the offset value to the beginning of the file.

**[Return value]**

When the file position is successfully changed, the new file position must be returned. If it is not successfully changed, -1L must be returned.

## 9.5.6 isatty Function

**The isatty function should be generated according to the specifications in this section.**
**int isatty(int fileno);**

■ **isatty Function**

**[Explanation]**

The file specified by fileno is to be checked to see whether it is a terminal file. When the file is a terminal file, true must be returned. If not, false must be returned.

**[Return value]**

When the specified file is a terminal file, true must be returned. If not, false must be returned.

# 9.5.7 sbrk Function

**The sbrk function should be generated according to the specifications in this section.**
**char *sbrk( int size);**

■ **sbrk Function**

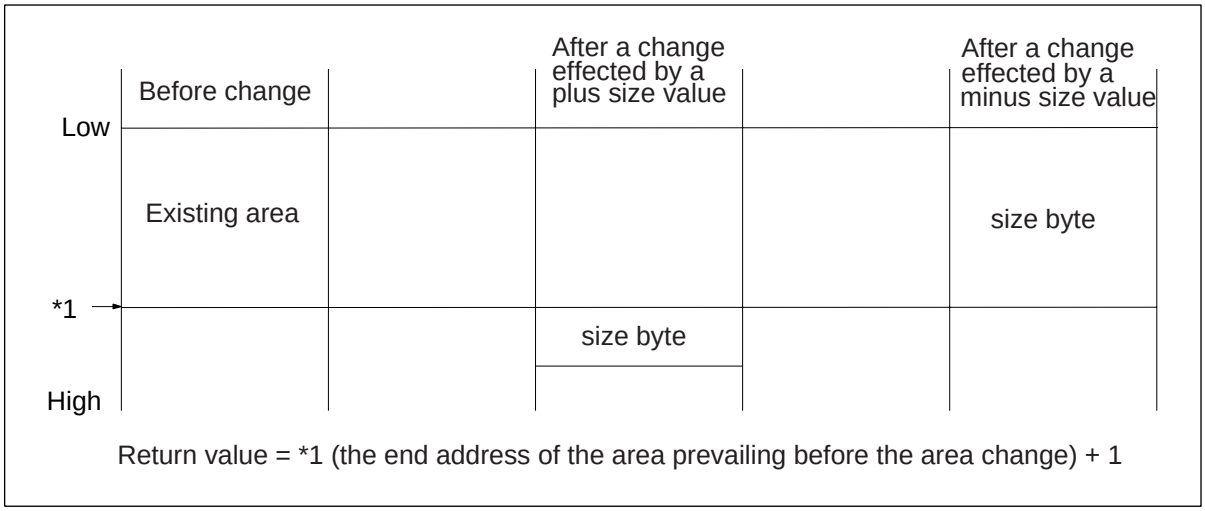
**[Explanation]**

The existing area must be enlarged by size bytes. If size is a negative quantity, the area must be reduced.

If the sbrk function has not been called, furnish a size-byte area.

The area varies as shown in Figure 9.5-1 "Area Change Brought About by sbrk Function Calling" by calling the sbrk function.

**Figure 9.5-1 Area Change Brought About by sbrk Function Calling**


Before change
After a change effected by a plus size value
After a change effected by a minus size value
Low
Existing area
size byte
*1
size byte
High
Return value = *1 (the end address of the area prevailing before the area change) + 1

**[Return value]**

When the area change is successfully made, the value to be returned must be determined by adding the value 1 to the end address of the area prevailing before the area change. If the sbrk function has not been called, the start address of the acquired area must be returned. If the area change is not successfully made, the value (char)-1 must be returned.

## 9.5.8 __exit Function

**The __exit function should be generated according to the specifications in this section.**

```
#include <stdlib.h>
void _exit (int status);
```

■ **__exit Function**

**[Explanation]**

The _exit function must bring the program to a normal end. When the status value is 0 or in the case of EXIT_SUCCESS, the successful end state must be returned to the system environment. In the case of EXIT_FAILURE, the unsuccessful end state must be returned to the system environment.

**[Return value]**

The _exit function does not return to the caller.

## 9.5.9 __abort Function

**The __abort function should be generated according to the specifications in this section.**
**void _abort( void);**

■ **__abort Function**

**[Explanation]**

The _abort function must bring the program to an abnormal end.

**[Return value]**

The _abort function does not return to the caller.

# CHAPTER 10 COMPILER-DEPENDENT SPECIFICATIONS

**This chapter describes the specifications that vary with the compiler. Descriptions are related to JIS standards that are created based on ANSI standards.**


10.1 "Compiler-dependent C Language Specification Differentials"

10.2 "Type of Floating-point Data and Range of Representable Values"

10.3 "Dissimilarities between C++ Specifications for C/C++ Compiler and ISO"

10.4 "C++ Specifications for C/C++ Compiler and EC++ Specifications"

10.5 "Limitations on Use of C++ Template"

# 10.1 Compiler-dependent C Language Specification Differentials

**Table 10.1-1 "Compiler-dependent Language Specification Differentials" lists the compiler-dependent C language specification differentials.**

## ■ Compiler-dependent Language Specification Differentials

**Table 10.1-1 Compiler-dependent Language Specification Differentials**

| Specification Differentials | Related Section in the JIS Standard | This Compiler |
|---|---|---|
| Japanese language process support and code system | 5.2.1 "Character sets"<br>6.1.2 "Identifiers" | No support<br>EUC or SJIS entries can be made only in the comment (can not be mixed) |
| Recognized character count of an identifier with an external binding | 6.1.2 "Identifiers" | No limitation |
| Differentiation between upper- and lower-case alphabetical characters of an identifier with an external binding | 6.1.2 "Identifiers" | Treated as different characters |
| Character set element expression code system | 6.1.3 "Numerical constants" | ASCII code |
| Char type treatment and expressible value range | 6.2.1.1 "Character type and integer type" | Unsigned(*1)<br>0 to 255 |
| Floating-point data formats and sizes<br>float type<br>double/long double type | 6.1.2.5 "Data types" | IEEE type(*2)<br>4 bytes<br>8 bytes |
| Whether or not to treat the start bit as signed bit when following types specified as bit field<br>char, short, int, long and long long type | 6.5.2.1, "Structure specifier and union specifier" | Not treated as a sign(*1) |
| Types that can be specified as bit field | 6.5.2.1, "Structure specifier and union specifier" | char type<br>signed char type<br>unsigned char type<br>short type<br>unsigned short type<br>int type<br>unsigned int type<br>long type<br>unsigned long type<br>long long type (fcc 935s only)<br>unsigned long long type (fcc 935s only) |

**Table 10.1-1 Compiler-dependent Language Specification Differentials (Continued)**

| Specification Differentials | Related Section in the JIS Standard | This Compiler |
|---|---|---|
| Structure or union member boundary alignment value<br>char type<br>signed char type<br>unsigned char type<br>short int type<br>unsigned short int type<br>int type<br>unsigned int type<br>long int type<br>unsigned long int type<br>long long int type<br>unsigned long long int type<br>float type<br>double type<br>long double type<br>Pointer type | 6.5.2.1, "Structure specifier and union specifier" | <br>1 byte<br>1 byte<br>1 byte<br>2 byte<br>2 byte<br>4 byte<br>4 byte<br>4 byte<br>4 byte<br>8 bytes for fcc935s command<br>8 bytes for fcc935s command<br>4 byte<br>8 byte<br>8 byte<br>4 byte |
| Character constant expression code system for preprocessor | | ASCII code |
| Registers that can be specified within asm statement | | R0, R1, R2, R3, R12, R13 for fcc911s command(*3)<br>GR4 to GR7, GR32 to GR47, FR0 to FR15, FR32 to FR47 for fcc935s command(*3) |
| ANSI-compliant standard library function support | | Refer to the volume entitled Libraries. |

*1: Alterable through option use.
*2: See section 11.2 "Low-level Function Library Use", Floating-point Data Format and Expressible Value Range.
*3: The other registers can be used when they are saved and recovered by the user.

# 10.2 Type of Floating-point Data and Range of Representable Values

**Table 10.2-1 "Type of Floating-point Data and Range of Representable Values" lists the floating-point data types and the range of values that can be expressed for each type.**

## ■ Floating-point Data Format and Expressible Value Range

**Table 10.2-1 Type of Floating-point Data and Range of Representable Values**

| Type of floating-point data | Range of Representable Values |
|---|---|
| float type | The exponent part is a value between 126 and + 127. (Base 2)<br>The fractional portion of the mantissa (the integer portion is normal ized to 1) is binary and has 24-digit accuracy.<br>The exponent part is value between 1022 and + 1023. (Base 2) |
| double type | The exponent part is a value between 1022 and + 1023. (Base 2)<br>The fractional part of the mantissa (the integer part is normalized to 1) is binary and has 53-digit accuracy. |
| long double type | The fractional part of the mantissa (the integer part is normalized to 1) is binary and has 53-digit accuracy. |

# 10.3 Dissimilarities between C++ Specifications for C/C++ Compiler and ISO

**This section explains the dissimilarities between the C++ specifications for the C/C++ compiler and ISO/IEC 14882:1998.**

## ■ Modifications to C++ Specifications for C/C++ Compiler from ISO

- Differences between the C++ specifications for the C/C++ compiler and ISO/IEC 14882:1998 are explained below.
- There is no support for the name-solving function in [temp.res] (Section 14.6) and [temp.dep] (Section 14.6.2) mentioned in the ISO/IEC 14882:1998 Specification.
- There is no support for the template argument function in [temp.arg.template] (Section 14.3.3) mentioned in the ISO/IEC 14882:1998 Specification.
- There is no support for the international character set function such as \uabcd.
- There is no support for the export keyword and export function.
- There is no support for the runtime type identification function including dynamic_cast and typeid operators.
- There is no support for the exception handling function including try-catch and throw.
- Only the C standard and EC++ libraries are provided. Libraries containing templates such as STL are not available.

# 10.4 C++ Specifications for C/C++ Compiler and EC++ Specifications

**This section describes the C++ specifications for the C/C++ compiler and the EC++ specifications.**

## ■ C++ Specifications for C/C++ Compiler and EC++ Specifications

At default, the C/C++ compiler operates to the EC++ specifications. A warning message is output at description of specifications outside the EC++ range. To delete just this warning, specify the -Ja option (including extended specifications) or the -Jc option (stringent specifications).

For details of the EC++ specifications, refer to http://www.caravan.net/ec2plus/.

# 10.5 Limitations on Use of C++ Template

**This section deals with the limitations on use of the C++ template.**

## ■ Limitations on Use of C++ Template

### ❍ Options specified when using template

Specify the -Ja option or the -Jc option. If this option is not given, description of specifications is outside the range of the EC++ specifications and the warning message is output.

### ❍ Limitations

When instantiation is not controlled by #pragma, or anything other than "local" (default: "none") is selected in the -t option, the following limitations apply:

- Linking is impossible just with the generated object file. The source files and simultaneously generated files with .ii extensions are needed in pairs.
- Do not move, delete or rewrite the generated files with .ii extensions.
- Do not change the directories and names of the generated object and source files and do not link object files after deletion. In this case, always perform recompiling. Recompiling is needed for changes alone between the UNIX and Windows environments.
- When the -S option is specified, always generate object files with .obj extensions with the same module name in the same directories as those of the generated assembler files.
- Data in the assembler files generated by the -S option is not rewritten.
- Do not generate libraries with a librarian.
- For development by more than one operator, carry out tasks from compiling to linking at a single point. Compiling at multiple points requires conformity in the positions of directories of source and object files (including the name of a network drive when used), the environment variables related to the C/C++ compiler, and the directory where the C/C++ compiler is installed.
- When compiling at multiple points in the UNIX environment, the true directory name not including a symbolic link must be the same.
- Linking is delayed depending on how to use a template.
- A warning of errors in the template syntax is issued at linking.

## ■ Circumventing limitations on the use of the C++ template

To circumvent the above problems, simply specify the -t local option for all the modules. This specification causes template data in each module, resulting in an increased code size for the runtime object.

To solve the code size problem, specify the --no_auto_instantiation option to embed #pragma in the source program, thereby controlling template instantiation.

For information on using #pragma to control instantiation, see section 6.12 "Function for Controlling Instantiation of C++ Template".

# CHAPTER 11 SIMULATOR DEBUGGER LOW-LEVEL FUNCTION LIBRARY

**The simulator debugger low-level function library is a library of the low-level functions which are necessary when the standard library is used with the simulator debugger. This chapter describes how to use the simulator debugger low-level function library.**


11.1 "Low-level Function Library Overview"

11.2 "Low-level Function Library Use"

11.3 "Low-level Func. Function"

11.4 "Low-level Function Library Change"

# 11.1 Low-level Function Library Overview

**The low-level function library is outlined below.**

## ■ Low-level Function Library Overview

The low-level function library offers the functions that are necessary when the standard library is used with the simulator debugger. The main functions are as follows.

- File manipulation functions based on I/O port simulation (open, close, read, write, lseek, isatty)
- Dynamic memory allocation function (sbrk)

In the simulator debugger, the program executed cannot terminate its own execution. Therefore, prepare the _abort and _exit functions.

## ■ File System Overview

The low-level function library uses the I/O port simulation function of the simulator debugger to carry out standard input/output operations and input/output operations relative to files. These operations are completed by performing input/output operations relative to one I/O port area which is regarded as one file.

When the open function is called, it allocates a 1-byte area of the I/O port simulation area (I/O section) defined by the low-level function library, and returns as the file number the offset from the beginning of the allocated area.

The read function and write function perform input/output operations relative to the 1-byte area allocated by the open function.

Input/output operations can be performed relative to the standard input/output and files when such standard input/output and files are allocated to the above-mentioned area prior to program execution using simulator debugger commands set inport and set outport.

The close function frees an already allocated area to render it reusable. Since the file position cannot be changed in the simulator debugger, the value -1 is always returned for the lseek function.

## ■ Area Management

An already acquired external variable area is used as the area returned by the sbrk function.

When the sbrk function is called, area allocation begins with the lowest address of the area.

# 11.2 Low-level Function Library Use

**This section describes the load module creation and simulator debugger setup procedures to be performed for low-level function library use.**

## ■ Initialization

No initialization is required except for _steam_init function calling.

When creating the startup routine in accordance with the system, call the _stream_init function prior to main function calling.

## ■ Load Module Creation

After completing creating of the necessary program, compile and link all the necessary modules. No special option specifying is needed.

The following libraries and startup routine are linked.

- startup.obj
- Standard library (lib911.lib)
- Low-level function library (lib911if.lib, lib935if.lib)

The sections are arranged at the following addresses.

- IOPORT: Address 0
- STACK: Address 0x100000
- Other: Address 0x1000

To change the IOPORT section arrangement, specify the -sc IOPORT=address option at compling. Describe the section arrangement address at the address position.

## ■ Simulator Debugger Setup

**[Setup for Standard Input/Output Use]**

```
set inport/ascii IOPORT, 0xff, $TERMINAL
set outport/ascii IOPORT+1, 0xff, $TERMINAL
```

Enter the address where the IOPORT section was positioned at linking in the above IOPORT position. If the -sc option is not specified at linking, the following results.

```
set inport/ascii 0, 0xff, $TERMINAL
set outport/ascii 1,  0xff, $TERMINAL
```

Since the first three areas of the IOPORT section are used for standard input, standard output, and standard error output, the other files are allocated to the fourth and subsequent areas (The offset from the beginning of the IOPORT section is 3).

In other words, allocation is performed sequentially in the order of file opening (offset 3, offset 4, etc.). Therefore, perform setup accordingly using the set inport and set outport commands.

To open a.doc as the input file and then open b.doc as the output file, setup as indicated below.

```
set inport/ascii IOPORT+3, h’ff, "a.doc"
```

```
set outport/ascii IOPORT+4, h'ff, "b.doc"
```

■ **Example**

Create a program that displays the character string "Hello!"! and initiate execution with the simulator debugger.

```
main( )
{
    printf("Hello!!\n");
}
```

Create a C-source file named test.c as indicated above.

Compile using the following command.

```
> fcc911s test.c -cpu MB91100
```

```
> fcc935s test.c -cpu MB93500
```

At completion of the preceding step, test.abs is created. Execute the created file with the simulator debugger.

After startup, input following commands.

```
> set inport/ascii h'0, h'ff, $TERMINAL
```

```
> set outport/ascii h'1, h'ff, $TERMINAL
```

```
> go, end
```

Since standard input is not involved in the above example, the set inport command can be omitted.

# 11.3 Low-level Func. Function

**This section describes the function specific to the simulator debugger low-level functions.**

## ■ Special I/O Port

As far as the low-level functions are concerned, the first three bytes of the I/O section are specified to function as the standard input, standard output, and standard error output, respectively. For such bytes, files No. 0, 1, and 2 are allocated. They are initialized to the opened state.

Table 11.3-1 "Predefined I/O Port" show the predefined I/O port.

**Table 11.3-1 Predefined I/O Port**

| Address | File Number | File Type |
|---|---|---|
| IOPORT | 0 | Standard input |
| IOPORT+1 | 1 | Standard output |
| IOPORT+2 | 2 | Standard error output |

The input from the standard input (file No. 0) is output to the standard output (file No. 1). The input to the standard input (file No. 0) is discontinued if the new line character \n is entered. However, when the input is fed from some other port, the input continues until the required number of characters are read.

## ■ open Function

The open function finds an unused I/O port area and then returns as the file number the area's offset from the beginning of the I/O section. In such an instance, the file name and open mode are not to be specified. Even if files are opened using the same file name, differing file numbers are assigned to them. Files No. 0, 1, and 2 are initialized to the opened state. Therefore, the open function begins allocation with file No. 3 unless files 0, 1, and 2 are subjected to the close process.

## ■ read Function

The read function reads data from the I/O port area specified by the address which is determined by adding the specified file number to the I/O section start address. The input from file No. 0 is treated as a line input. When the new line character \n is entered, the read function terminates even if the required character count is not reached. Further, this input is output to the standard output (file No. 1). The input from a file numbered other than 0 is treated as a block input. Reading continues until the required character count is reached.

## ■ write Function

The write function writes data to the I/O port area specified by the address which is determined by adding the specified file number to the I/O section start address. Unlike the input, the operation does not vary with the I/O port area address

## ■ lseek Function

The file position cannot be specified in the simulator debugger. Therefore, the value -1, which indicates an unsuccessful file position change, is always returned.

### ■ isatty Function

In the case of file No. 0, 1, or 2, true is returned. In the other cases, false is returned.

### ■ close Function

The close funciton releases the port related to the specified file number.

### ■ sbrk Function

The simulator debugger does not provide a means of dynamic memory allocation. Therefore, the sbrk func-tion acquires a fixed area and uses it. To change the area or its size, create an alternative function and substitute it for the sbrk function with a librarian. For details, see section 11.4 "Low-level Function Library Change".

# 11.4 Low-level Function Library Change

**This section shows how to change the dynamic allocation area.**

## ■ Dynamic Allocation Area Change

Locate the following line in the sbrk.c source program list. Change the value in this line to the dynamic allocation area size (in bytes).

```
#define  HEEP  SIZE          16*1024
```

Use the following commands to compile and update the library.

```
> fcc911s  -O  -c  sbrk.c  -cpu  MB91100
> flib911s  -r  sbrk.obj  lib911if.lib  -cpu  MB91100
> fcc935s  -O  -c  sbrk.c  -cpu  MB93500
> flib935s  -r  sbrk.obj  lib935if.lib  -cpu  MB93500
```

When the above change is made, the dynamic allocation area is secured as the sbrk.c static external variable without being positioned at the beginning of the stack.

## ■ fcc911s Command sbrk.c Source Program List

The source program required for changing the dynamic area is shown below. The file name must be sbrk.c.

```
#define     HEEP_SIZE           16*1024
static      long                brk_siz  =  0;
#if         HEEP_SIZE
                                        typedef int         _heep_t;
#define ROUNDUP(s)              (((s)+sizeof(_heep_t)-1)&~(sizeof(_heep_t)-1))
static      _heep_t             _heep[ROUNDUP(HEEP_SIZE)/sizeof(_heep_t)];
#define                         _heep_size              ROUNDUP(HEEP_SIZE)
#else
extern      char                *_heep;
extern      long                _heep_size;
#endif

extern      char      *sbrk(int size)
{
    if (brk_siz + size >  _heep_size || brk_siz + size < 0)
            return((char*)-1);
    brk_siz += size;
    return( (char *)_heep + brk_siz - size);
}
```

## ■ fcc935s Command sbrk.c Source Program List

The source program required for changing the dynamic area is shown below. The file name must be sbrk.c.

```
#define     HEEP_SIZE           16*1024
static      long                brk_siz  =  0;
#if       HEEP_SIZE
typedef     long long           _heep_t;
#define ROUNDUP(s)              (((s)+sizeof(_heep_t)-1)&~(sizeof(_heep_t)-1))
static      _heep_t             _heep[ROUNDUP(HEEP_SIZE)/sizeof(_heep_t)];
#define                         _heep_size          ROUNDUP(HEEP_SIZE)
#else
extern      char                *_heep;
extern      long                _heep_size;
#endif

extern      char                *sbrk(int size)
{
   if (brk_siz + size >  _heep_size || brk_siz + size < 0)
            return((char*)-1);
   brk_siz += size;
   return( (char *)_heep + brk_siz - size);
}
```

# APPENDIX

**The appendix gives a list of types, macros, functions and variables provided by the libraries and describes the operations specific to the libraries.**

APPENDIX A "List of Types, Macros, Functions, and Variables Provided by C Libraries"

APPENDIX B "Operations Specific to C Libraries"

# APPENDIX A List of Types, Macros, Functions, and Variables Provided by C Libraries

**The types, macros, functions, and variables provided by the C libraries are listed below.**

## ■ assert.h

### ❍ Function

assert

## ■ ctype.h

### ❍ Macros

| | | | | |
|---|---|---|---|---|
| isalnum | isalpha | iscntrl | isdigit | isgraph |
| islower | isprint | ispunct | isspace | isupper |
| isxdigit | tolower | toupper | | |

## ■ errno.h

### ❍ Macros

EDOM ERANGE

### ❍ Variable

errno

## ■ float.h

### ❍ Macros

| | | | |
|---|---|---|---|
| FLT_RADIX | FLT_ROUNDS | FLT_MANT_DIG | DBL_MANT_DIG |
| LDBL_NANT_DIG | FLT_DIG | DBL_DIG | LDBL_DIG |
| FLT_MIN_EXP | DBL_MIN_EXP | LDBL_MIN_EXP | FLT_MIN_10_EXP |
| DBL_MIN_10_EXP | LDBL_MIN_10_EXP | FLT_MAX_EXP | DBL_MAX_EXP |
| LDBL_MAX_EXP | FLT_MAX_10_EXP | DBL_MAX_10_EXP | LDBL_MAX_10_EXP |
| FLT_MAX | DBL_MAX | LDBL_MAX | FLT_EPSILON |
| DBL_EPSILON | LDBL_EPSILON | FLT_MIN | DBL_MIN |
| LDBL_MIN | | | |

## ■ limits.h (for fcc911s command)

### ❍ Macros

| | | | | |
|---|---|---|---|---|
| MB_LEN_MAX | CHAR_BIT | SCHAR_MIN | SCHAR_MAX | UCHAR_MAX |
| CHAR_MIN | CHAR_MAX | INT_MIN | INT_MAX | UINT_MAX |
| SHRT_MIN | SHRT_MAX | USHRT_MAX | LONG_MIN | LONG_MAX |
| ULONG_MAX | | | | |

## ■ limits.h (for fcc935s Command)

### ❍ Macros

| | | | | |
|---|---|---|---|---|
| MB_LEN_MAX | CHAR_BIT | SCHAR_MIN | SCHAR_MAX | UCHAR_MAX |
| CHAR_MIN | CHAR_MAX | INT_MIN | INT_MAX | UINT_MAX |
| SHRT_MIN | SHRT_MAX | USHRT_MAX | LONG_MIN | LONG_MAX |
| ULONG_MAX | LONG_LONG_MIN | LONG_LONG_MAX | ULONG_LONG_MAX | |

## ■ math.h

### ❍ Macros

HUGE_VAL EDOM ERANGE

### ❍ Function

| | | | | |
|---|---|---|---|---|
| acos | asin | atan | atan2 | cos |
| sin | tan | cosh | sinh | tanh |
| exp | frexp | ldexp | log | log10 |
| modf | pow | sqrt | ceil | fabs |
| floor | fmod | | | |

## ■ setjmp.h

### ❍ Type

jmp_buf

### ❍ Macros

setjmp

### ❍ Function

longjmp

## ■ stdarg.h

### ❍ Type

va_list

### ❍ Macros

va_start va_arg va_end

## ■ stddef.h

### ❍ Type

ptrdiff_t size_t

### ❍ Macros

NULL offsetof

## ■ stdio.h

### ❍ Type

| | | | |
|---|---|---|---|
| ptrdiff_t | size_t | FILE | fpos_t |

### ❍ Macros

| | | | | |
|---|---|---|---|---|
| NULL | EOF | SEEK_SET | SEEK_CUR | SEEK_END |
| _IONBF | _IOLBF | _IOFBF | BUFSIZ | stdin |
| stdout | stderr | putchar | putc | getchar |
| getc | offsetof | | | |

### ❍ Function

| | | | | |
|---|---|---|---|---|
| putchar | putc | getchar | getc | fclose |
| fflush | fopen | freopen | setbuf | setvbuf |
| fprintf | fscanf | printf | scanf | sprintf |
| sscanf | vfprintf | vprintf | vsprintf | fgetc |
| fgets | fputc | fputs | gets | puts |
| ungetc | fread | fwrite | fgetpos | fseek |
| fsetpos | ftell | rewind | clearerr | feof |
| ferror | | | | |

## ■ stdlib.h

### ❍ Type

| | | | |
|---|---|---|---|
| ptrdiff_t | size_t | div_t | ldiv_t |

### ❍ Macros

| | | | | |
|---|---|---|---|---|
| NULL | offsetof | EXIT_FAILURE | EXIT_SUCCESS | RAND_MAX |

❍ **Function**

| | | | | |
|---|---|---|---|---|
| atof | atoi | atol | strtod | strtol |
| strtoul | rand | srand | calloc | free |
| malloc | realloc | abort | atexit | exit |
| besearch | qsort | abs | div | labs |
| ldiv | | | | |

■ **string.h**

❍ **Type**

ptrdiff_t size_t

❍ **Macros**

NULL offsetof

❍ **Function**

| | | | | |
|---|---|---|---|---|
| memcpy | memmove | strcpx | strncpy | strcat |
| strncat | memcmp | strcmp | strncmp | memchr |
| strchr | strcspn | strpbrk | strrchr | strspn |
| strstr | strtok | memset | strlen | |

■ **fcntl.h**

❍ **Macros**

| | | | | |
|---|---|---|---|---|
| O_RDONLY | O_WRONLY | O_RDWR | O_APPEND | O_CREAT |
| O_TRUNC | O_BINARY | | | |

■ **unistd.h**

❍ **Macros**

SEEK_SET SEEK_CUR SEEK_END

## ■ sys/types.h

### ❍ Type

off_t

# APPENDIX B Operations Specific to C Libraries

**The operations specific to the C libraries are described below.**

## ■ Operations Specific to C Libraries

❍ **Diagnostic information printed out by the assert function and assert function termination operation**

**[Diagnostic information]**

< Program Diagnosis *** information of fail expression >

file: File name expanded by __FILE__

line: Line number expanded by __LINE__

expression: Expression

**[Termination operation]**

Same as the abort function calling.

❍ **Inspection character sets for isalnum, isalpha, iscntrl, islower, isprint, and isupper functions**

- isalnum: '0' to '9', 'a' to 'z', or 'A' to 'Z'
- isalpha: 'a' to 'z' or 'A' to 'Z'
- iscntrl: '\100' to '\037', or '\177'
- islower: 'a' to 'z'
- isprint: '\040' to '\176'
- isupper: 'A' to 'Z'

❍ **Mathematical function return value upon definition area error occurrence**

- q N a N

❍ **Whether the mathematical function sets up the macro ERANGE value for errno upon underflow condition occurrence**

- ERANGE
- The detectable result value must be +0 or -0.
- The undetectable result value is undefined. It depends on the function.

❍ **When the second actual argument for the fmod function is 0**

The definition area error must occur or the value 0 must be returned.

❍ **File buffering characteristics**

**[Input file buffering characteristics]**

- IOLBF, IOFBF: Full buffering

- IONBF: No buffering

**[Output file buffering characteristics]**

- IOFBF: Full buffering
- IOLBF: Line buffering
- IONBF: No buffering.

**[Full buffering]**

Buffering is conducted using all the preset buffer areas. When the input function is called at the time of input from a file, any data remaining in the buffer is returned as the input from the file. If the buffer is emptied of data or does not have sufficient data, the input from the file is received until the buffer is filled up and then only the necessary amount is returned as the input. At the time of output to a file, the output function writes into the buffer instead of outputting into the file. When the buffer is filled up by the write operation, the buffer outputs its entire contents to the file.

**[Line buffering]**

Buffering is conducted for each output line.

**[No buffering]**

File input/output is implemented in compliance with the input/output request made by input/output function calling. Unlike the other buffering operations, no data will be saved into the memory.

❍ **%p format conversion output for fprintf function**

- The number of digits is less than 8 at 8-digit hexadecimal notation, preceding 0s are padded. Uppercase alphabetical characters are used.

❍ **%p format conversion input format for fscanf function**

If less then 8 digits in hexadecimal notation are used (in upper- or lower-case alphabetic characters), the leading 0s are padded. If the specified number of digits is exceeded, only the low-order part is valid.

❍ **Conversion of long long type value of fprintf function for fcc935s command**

To convert a long long type value, specify 11 or q immediately before the format converter. The same conversion result is obtained regardless of whether 11 or q is specified. 11 or q can be specified for conversion specifications %d, %i, %o, %u, %x, and %X.

**[Example]**

```
long long a = 10LL:

fprintf(stdout, "%lld\n", a);
```

❍ **Conversion of long long type value of fscanf function for fcc935s command**

To convert a long long type integer value, specify 11 or q immediately before the format converter. The same conversion result is obtained regardless of whether 11 or q is specified. 11 or q can be specified for conversion specifications %d, %i, %o, %u, and %x.

**[Example]**

```
long long a, b:

fscanf(stdin, "%lld%lln", &a, &b);
```

❍ **Interpretation of a single "-" character appearing at a position other than the start and end**

**of the scan-list relative to %[ format conversion**

A string of consecutive characters beginning with the character placed to the left of "-" and ending with the character placed to the right of "-" is handled.

**[Example]**

%[a-c] is equal to [abc].

❍ **abort function operation relative to an open file**

Closing takes place after flushing of all streams.

❍ **Status returned by the exit function when the actual argument value is other than 0, EXIT_SUCCESS, and EXIT_FAILURE**

The status to be returned is the same as for EXIT FAILURE.

❍ **Floating-point number limit values**

| | | | | |
|---|---|---|---|---|
| FLT_MAX | 7F7F | FFFF | | |
| DBL_MAX | 7FEF | FFFF | FFFF | FFFF |
| FLT_EPSILON | 3400 | 0000 | | |
| DBL_EPSILON | 3CB0 | 0000 | 0000 | 0000 |
| FLT_MIN | 0080 | 0000 | | |
| DBL_MIN | 0010 | 0000 | 0000 | |

❍ **Limitations on setjmp function**

The interrupt environment is not supported by the libraries. Therefore, the interrupt handler cannot achieve environment saving and the return to the interrupt handler cannot be made.

❍ **Limitations on va_start macro**

Do not use the following variable definitions for va_start macro first argument.

- char type, unsigned char type, short type, or unsigned short type (however, the pointer type for these types can be used)
- Type having the register storage area class.
- Function type
- Array type
- Type different from the type derived from existing actual argument extension.

❍ **File types**

Files that can be handled by the libraries are divided into two types; text files and binary files. The libraries treat the text files and binary files in the same manner except for the difference in the second argument of the open function called upon file opening. When a binary file is specified, O_BINARY is added to the second argument of the open function. For the open function argument, see section 9.5.1 "open Function".

❍ **div_t type and ldiv_t type**

It is equivalent to an undermentioned structural body.

```
div_t:      struct  {
                int     quot;
                int     rem;
            }:
ldiv_t:     struct      {
              long  int   quot;
              long  int   rem;
            };
```

- **abort function operations**

  When the abort function is called, all the open output streams are flushed and then all the open streams are closed. Finally, the _abort function is called.

- **Maximum count of functions that can be registered by the atexit function**

  Up to 20 functions can be registered.

- **exit function operations**

  When the exit function is called, all the functions registered by the atexit function are called in the reverse order of registration, all the open output streams are flushed, and then all the open streams are closed. Finally, the _exit function is called with the status value, which is delivered as the argument, retained. When the status value is 0 or EXIT_SUCCESS, it indicates successful termination. When the status val-ue is EXIT_FAILURE, it indicates the unsuccessful termination.

# INDEX

**The index follows on the next page.**
**This is listed in alphabetic order.**

# Index


**A**

abort function ..... 181
addition operation (halfword dual) instruction (M instruction) ..... 140, 143
addition sum operation (halfword quad) instruction (M instruction) ..... 143
ANSI standard, macro stipulated by ..... 151
area management ..... 192
asm statement, description by ..... 118
assert.h ..... 200

**B**

branch probability specification function ..... 132

**C**

C library, operation specific to ..... 206
C++ specification for C/C++ compiler and EC++ specification ..... 188
C++ template, limitation on use of ..... 189
C/C++ compiler function ..... 2
case sensitiveness ..... 19
close function ..... 174, 196
command basic process ..... 3
command cancel option, list of ..... 24
command line ..... 16
command line, position within ..... 19
command operand ..... 17
command option, list of ..... 20
command process ..... 16
command related option ..... 27, 64
compiler translation, limitation on ..... 153
compiler-dependent language specification differential ..... 184
compiling process control, option for ..... 3
controlling instantiation of C++ template, function for ..... 156
coordination with symbolic debugger ..... 5
ctype.h ..... 200

**D**

data output related option ..... 27, 34
debug information related option ..... 27, 63
default option file ..... 69
dependency ..... 19
directry name ..... 18
dynamic alocation area change ..... 197

**E**

errno.h ..... 200
error level ..... 70
example ..... 194
exclusiveness ..... 19
execution process overview ..... 158
exit function ..... 180
expressible value range ..... 186

**F**

fcc907s command section structure ..... 74
fcc911 command, macro predefined by ..... 151
fcc911, rule for name generation ..... 76
fcc911s command argument ..... 85
fcc911s command argument extension format ..... 88
fcc911s command bit field ..... 78
fcc911s command boundary alignment ..... 77
fcc911s command calling procedure ..... 89
fcc911s command function call interface ..... 82
fcc911s command interrupt function call interface ..... 93
fcc911s command library section name ..... 164
fcc911s command register guarantee ..... 91
fcc911s command register setup ..... 91
fcc911s command return value ..... 92
fcc911s command sbrk.c source program list ..... 197
fcc911s command stack frame ..... 83
fcc911s command structure/union ..... 80
fcc935, generation rules for names used by ..... 100
fcc935s command argument ..... 109
fcc935s command argument extension format ..... 111
fcc935s command bit field ..... 102
fcc935s command boundary alignment ..... 101
fcc935s command calling procedure ..... 112
fcc935s command function call interface ..... 106
fcc935s command library section name ..... 164
fcc935s command register ..... 114
fcc935s command register setup ..... 114
fcc935s command sbrk.c source program list ..... 198
fcc935s command section structure ..... 98
fcc935s command stack frame ..... 107
fcc935s command structure/Union ..... 104

fcc935s command, macro predefined ..... 152
fcntl.h ..... 204
FELANG ..... 13
FETOOL ..... 8
file name ..... 18
file type ..... 164
file-system overview ..... 192
float.h ..... 201
floating-point data format ..... 186

**G**

generation rule for name used by fcc935s ..... 100

**H**

header file search ..... 5

**I**

I/O area access function ..... 127
INC911/INC935 ..... 11
initialization ..... 193
initialization/termination process ..... 169
in-line expansion specifying function ..... 129
integer operation built-in function ..... 150
interrupt function calling procedure ..... 95
interrupt function description function ..... 125
interrupt level setup function ..... 125, 131
interrupt mask disable function ..... 124
interrupt mask setup function ..... 124
interrupt stack frame ..... 94
interrupt vector table generation function ..... 125
isatty function ..... 178, 196
iseek function ..... 177, 195

**L**

language specification related option ..... 27, 40
LIB911/LIB935 ..... 9
library use, process and function must be prepared for ..... 168
limitation on compiler translation ..... 153
limitations on use of C++ template ..... 189
limits.h (for fcc911s command) ..... 201
limits.h (for fcc935s command) ..... 201
linkage related option ..... 27, 65
low-level function library overview ..... 192
low-level function specification ..... 172
low-level function type ..... 166, 170

**M**

math.h ..... 202
media accumulator clear instruction (M instruction) ..... 148
media accumulator read ..... 148
media byte-halfword conversion instruction (M instruction) ..... 147
media custom instruction (M instruction) ..... 149
media CUT instruction (M instruction) ..... 145
media dual complex multiplication ..... 143
media dual cross multiplication (halfword dual) instruction (M instruction) ..... 139
media dual multiplication ..... 140
media dual multiplication (halfword dual) instruction (M instruction) ..... 139
media dual saturation addition ..... 138
media halfword expand instruction (M instruction) ..... 146
media halfword pack/unpack (dual) instruction (M instruction) ..... 147
media halfword pack/unpack instruction (M instruction) ..... 146
media instruction built-in function ..... 133
media logical operation (word) instruction (M instruction) ..... 135
media mean operation (halfword dual) instruction (M instruction) ..... 137
media multiplication ..... 143
media quad complex product sum operation (halfword quad) instruction (M instruction) ..... 144
media quad cross multiplication (halfword quad) instruction (M instruction) ..... 142
media quad multiplication (halfword quad) instruction (M instruction) ..... 142
media quad saturation addition ..... 141
media rotate (word) instruction (M instruction) ..... 136
media saturation operation (halfword dual) instruction (M instruction) ..... 138
media shift (halfword dual) instruction (M instruction) ..... 137
media trap instruction (M instruction) ..... 149
media word CUT instruction (M instruction) ..... 136
modification to C++ specification for C/C++ compiler from ISO ..... 187

**O**

open function ..... 173, 195
OPT911/OPT935 ..... 10
optimization ..... 6
optimization related option ..... 27, 46

option file ..... 68
option file limitation ..... 68
option file related option ..... 27, 67
option file, acceptable comment entry in ..... 68
option syntax ..... 19
output object related option ..... 27, 56

**P**

pragma instruction, description by ..... 121
preprocessing related option ..... 31
preprocessor related option ..... 27
process ..... 168

**R**

read function ..... 175, 195
re-include prevention function ..... 155
required process/low-level function ..... 171
rule for name generation with the fcc911 ..... 76

**S**

same option, multiple specifying of ..... 19
sbrk function ..... 179, 196
section name change function ..... 130
setjmp.h ..... 202
simulator debugger setup ..... 193
special I/O port ..... 195
standard error output file close ..... 169
standard error output file open ..... 169
standard input/output ..... 169
standard library function ..... 171
startup routine creation ..... 160
stdarg.h ..... 202
stddef.h ..... 202
stdio.h ..... 203
stdlib.h ..... 203
stream area initialization ..... 169
string.h ..... 204
subtraction (halfword dual) instruction (M instruction) ..... 138
subtraction (halfword quad) instruction (M instruction) ..... 141
subtraction operation (halfword dual) instruction (M instruction) ..... 140
symbolic debugger, coordination with ..... 5
sys/types.h ..... 205
system-dependent process ..... 166

**T**

TMP ..... 12
tool identifier ..... 70
translation control related option ..... 27, 28
translation process,message generated in ..... 70

**U**

unistd.h ..... 204

**W**

wait_nop built-in function ..... 133
write function ..... 176, 195
write instruction (M instruction) ..... 148

CM81-00205-2E

**FUJITSU SEMICONDUCTOR** • CONTROLLER MANUAL

FR-V/FR FAMILY

SOFTUNE C/C++ COMPILER MANUAL

for V5

January 2001 the second edition

Published **FUJITSU LIMITED** Electronic Devices

Edited Technical Communication Dept.